An Industrial Case Study on Speeding up User Acceptance Testing by Mining Execution Logs

Zhen Ming Jiang
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, ON, Canada
zmjiang@cs.queensu.ca

Alberto Avritzer
Software Development Technologies Group
Siemens Corporate Research
Princeton, NJ, USA
alberto.avritzer@siemens.com

Emad Shihab, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, ON, Canada
{emads, ahmed}@cs.queensu.ca

Parminder Flora
Enterprise Performance Engineering
Research In Motion
Waterloo, ON, Canada

***Abstract*—Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions. To determine whether the reliability of an application satisfies the reliability requirements, User Acceptance Testing is performed at deployment sites. To support the wide variation in configurations and usage patterns, User Acceptance Testing has become a crucial step in large deployments of mission-critical applications. However, verifying the long-term reliability of an application requires lengthy on-site engagements and dedicated use of costly lab setups.**

In this paper, we propose a technique to reduce the time and cost needed for User Acceptance Testing. We use a repository of execution logs from related deployments and prior tests of the application to mine reliability estimates. We then customize these estimates by mining logs generated from a limited-time User Acceptance Test (i.e., one day of testing) instead of from traditionally longer tests (e.g., one week of testing). Deployers of applications can use such customized estimates to determine whether an application satisfies their reliability requirements. Through a case study on a large-scale enterprise application, we show that our reliability estimate lies within 4% of the reliability estimate derived from the longer User Acceptance Tests.

I. Introduction

Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions [26]. With studies showing that many field reliability problems are not due to feature bugs, but rather due to applications not scaling to field workloads [16], [30], the reliability under field load is rapidly becoming an important concern for deployers of large mission-critical applications. Such applications range from web applications to telecommunication infrastructures, and they must support concurrent access by thousands or millions of users while functioning over a long period of time.

Reliability estimates must be produced whenever a new version of the mission-critical application is released. These estimates help customers determine whether an application meets their reliability requirements. Such reliability estimates are derived based on workloads from synthetic benchmark load runs and early field deployments of the application. These benchmark workloads rarely match the actual field workload, leading to estimates that do not match the expected field reliability of the application [13], [29]. This means that the general reliability estimates might not be realistic, i.e. not reflective of the actual field reliability [5].

Instead of relying on the reliability estimates of the application builders, nowadays, many large enterprise customers conduct their own in-house reliability verification before upgrading to new versions of mission-critical applications. This type of testing is commonly referred to as *User Acceptance Testing*. User Acceptance Testing entails customers deploying the new version on in-house labs with similar configurations to the production environments and simulating their expected workloads. For example, if the users in a particular deployment won't be using any of its buggy (not-as-reliable) features, then the expected reliability of the application will be much higher than the estimate provided by the general reliability estimate. However, the acceptance testing process is time-consuming and costly, requiring lengthy on-site engagements [18] – often leading customers to either reduce their acceptance testing efforts or to delay upgrading as they perform detailed acceptance testing. This state of practice prolongs the upgrade cycle for products and reduces the revenue stream of rapidly-growing companies that develop these systems.

To deal with the lengthy on-site engagements and costly lab time needed to perform User Acceptance Testing, we propose an approach in this paper to reduce the time needed for such on-site engagements. Instead of conducting the testing on-site over a full work week, we show that our approach can reduce the time needed to perform the testing down to a single day, while providing reliability estimates that are within 4% of the ones derived from a full User Acceptance Testing cycle. Our approach analyzes rarely used, yet readily available execution logs to produce empirically validated and customized reliabil-

ity estimates for mission-critical applications. Basically, our approach uses logs generated from a single day of acceptance testing to identify the usage patterns ("system states") that are important for a customer, and the "occurrence probability". Then, our approach uses logs derived from thousands of hours of execution from prior deployments and earlier tests of the same version of the application to provide a good estimate of how often the identified system states fail ("failure probabilities"). By combining the occurrence probabilities from User Acceptance Testing with the failure probabilities from log repositories, we can produce an accurate, customized reliability estimate using less resources and time.

The main contributions of our approach are as follows:

- Our approach requires no additional instrumentation or profiling, instead it leverages widely available, yet rarely used execution logs.
- Our approach reduces the time needed to perform User Acceptance Testing with high accuracy. This leads to reduced costs since on-site engagements can be shorter and the use of costly testing labs can be reduced.

Organization of the Paper

The remainder of this paper is organized as follows: Section II provides an example to motivate the limitations of traditional reliability estimates and the benefits of User Acceptance Testing and deployment-specific reliability estimates. Section III first presents an overview of our approach then describes the three phases (sections III-A, III-B, and III-C) involved in our reliability estimation approach. Section IV evaluates our approach on a large mission-critical application. Section V discusses our current approach and presents future work. Section VI presents the threats to validity. Section VII presents related work. Section VIII concludes this paper.

II. Motivating Example

We first present an example to motivate our approach. Jack's company is planning an upgrade to a new version of a mission-critical application. The application is an online web application with a web server for handling client requests and a database backend for storing transaction information. Jack must determine whether the new version of the application is reliable enough to warrant upgrading to it. Since the application is a critical application within Jack's organization, Jack requires the new version of the application to have a reliability of at least 0.99. A reliability of 1 means that the application never fails. A reliability of 0.99 indicates that 99% of the time the application will function correctly with no crashes and no Service Level Agreement (SLA) violations.

Tom, one of the developers of the application, informs Jack that the reliability of the new version is 0.99, based on Tom's in-house testing using synthetic industry-standard benchmarks. Due to the importance of the application, Jack wants a more custom reliability estimate that factors in his deployment environment and usage characteristics (i.e. workload). Jack requests Tom to dispatch an on-site engagement team to perform a full week of User Acceptance Testing on the new version in Jack's labs. Since the testing requires access to a similar lab setup as Jack's production setup, Jack would have to migrate some of his production load to other machines to free up machines for the full week of testing.

TABLE I: System State and Failure Profile Derived from Synthetic Runs and Other Deployments

System States (Search, Browse, Purchase)	Occurrence Probability	Failure Probability
(0,0,0)	0.40	0
(0,1,0)	0.30	0
(1,1,0)	0.20	0
(1,1,1)	0.10	0.10

TABLE II: System State Profile Recovered from a Day of User Acceptance Testing

System States (Search, Browse, Purchase)	Occurrence Probability
(0,0,0)	0.15
(0,1,0)	0.15
(1,1,0)	0.10
(1,1,1)	0.60

Tom explains to Jack that they can perform the whole User Acceptance Testing process within a single day instead of a longer time period (e.g. usually a full week). Jack is excited since this process could be done possibly over the weekend, when the production workload is low, so he does not have to book longer lab time, and the cost of the on-site engagement can be minimal. Tom proceeds to explain his technique to calculate the reliability of the application. His approach is based on work done by Avritzer [11] to model the reliability of telecommunication applications. Instead of simply capturing the workload using a black-box approach (e.g., by measuring metrics like the number of transactions per second), Tom uses a white-box approach that captures the internal state of the application as it processes the workload. These states are influenced by the deployment environment and workload. Tom defines the system state for his application as a 3-value tuple that captures the usage of the system in terms of the following three currently executing scenarios: browsing, purchasing and searching. Tom opts for this simple high-level definition, although other more complex low-level definitions are also possible. For example, the browsing scenarios could be further divided into browsing catalogs and browsing recommendations.

Using this system state model, Tom samples the execution of the application at run-time and determines which states it resides in. He defines each value in the tuple to be the number of active scenarios at the moment. For example, the system state (0, 0, 0) indicates the system is in the idle state. The state (1, 1, 1) indicates that the system is currently processing 1 search, 1 browse and 1 purchase scenario concurrently. Other models, such as those including the percentage of utilization, are possible as well.

By continuously sampling the data from the in-house testing and other deployments, Tom derives a system state profile for

the application, i.e., an overview of the system states that occur together with their frequency. Table I shows the system state profile derived from synthetic runs and other deployments. The profile indicates that the application is in idle state (0,0,0) 40% of the time. At each sample, Tom also determines if there are any reported failures of the application (e.g., crashes or SLA violations). He can then calculate the reliability for each system state. Looking back at Table I, we find that no failures are reported in the occurrences of state (0,0,0), while 10% of the occurrences of state (1,1,1) exhibit some type of failure. Using the sampled failure distribution and the system state profile, Tom can derive a reliability estimate for the application. By combining data from multiple deployments, Tom concludes that based on the lab testing and other deployments, the general reliability for this application is 0.99 (failure occurs only in $0.40 * 0 + 0.30 * 0 + 0.20 * 0 + 0.10 * 0.10 = 1\%$ of the cases).

However, this in-house reliability estimate does not consider the differences in usage and deployment patterns in different deployments. For example, after a day's worth of running the application at Jack's site using his workload, Tom notices that state (1,1,1) occurs at a much higher frequency (60% in Table II showing the occurrence probability of Jack's system states) compared to the data used to calculate the in-house reliability estimate (10% in Table I). It is clear that Jack's deployment spends more time in state (1,1,1), which has a high failure probability (based on in-house testing). This knowledge should be used to customize the reported in-house reliability estimate. The customized reliability estimate for Jack's deployment is 0.94 (failure occurs in $0.15 * 0 + 0.15 * 0 + 0.10 * 0 + 0.60 * 0.10 = 6\%$ of the cases) instead of the general in-house estimate of 0.99. Based on our customized reliability calculation, Jack should not deploy the new version of the application. This deployment estimate is derived from the occurrence probabilities of states in a single day of testing (second column of Table II), and the failure probabilities of states in hundreds of hours of execution in the lab and other deployments (third column of Table I).

We note three novel contributions of our approach:

1) It is important to capture the internal system states and their failure probability when we estimate the application reliability. The system state profile is influenced by both the usage pattern and the deployment characteristics. Given two deployments with the same usage pattern, they might still end up with different system state profiles because of different deployment characteristics. For example, if one of the deployments has a much slower database, the application might not be able to process searches as fast, causing the system states to have a higher number of active "Search" requests. The deployment with a slower database will likely have lower reliability even under the same workload.
2) Our approach makes use of the execution logs of the application to calculate the system state profiles, instead of instrumenting or profiling the application during runtime. Sampling an application during runtime is not feasible in a production setting due to the high overhead [12], [22]. Execution logs are readily available and are often used for remote issue resolution and for legal compliance purposes (e.g., "Sarbanes-Oxley Act of 2002" [3]). By sampling the logs at a constant frequency (e.g., once a second), we can create a system state profile and failure profile without impacting the performance of an application.
3) The use of execution logs permits Tom to continuously improve the reliability estimate of his application, since he can keep on integrating new logs coming from field deployments in an automated fashion. As for Jack, he would need to provide a log that captures his expected system profile. To provide such a log, Jack could provide logs from previous versions of the application or ideally provide logs from a limited deployment of the new version of the application. For example, Jack can provide the data for running the application for a single day and Tom can give him a reliability estimate based on hundreds of deployments that have been running over the past six months.

III. Approach Overview

Figure 1 gives an overview of our approach, which consists of the following three phases:

1) *Log Analysis:* Recover the executed scenario instances and identify the reported errors from execution logs.
2) *System State Derivation:* Derive the system states and calculate the occurrence probability and failure probability for each system state.
3) *Reliability Estimation:* Estimate the deployment-specific reliability using Bayesian Networks.

The *Log Analysis* and *System State Derivation* phases both analyze two sources of execution logs:

- An *execution log repository*, which stores the load test and other field deployment logs for the new version of the application. We analyze the execution log repository to get a collection of all possible system states as well as their *failure probability*.
- Sample *User Acceptance Testing logs*, which are execution logs generated during User Acceptance Tests at the targeted deployment site. We analyze these logs to obtain the *occurrence probability* distribution of those system states that occur in practice at the deployment site.

Then, we calculate the deployment-specific reliability by matching the occurrence probability distribution of the system states obtained from the User Acceptance Testing logs with the corresponding failure probability distribution of these system states obtained from the execution log repository.

In the next three subsections, we will use logs from a small online bookstore as a running example to explain the aforementioned three steps of our approach.

A. Log Analysis

Instead of instrumenting an application, we make use of the readily available execution logs to recover the needed

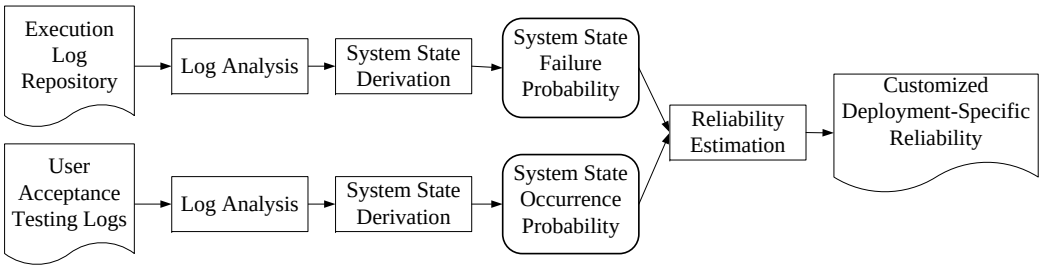

Fig. 1: An Overview of Our Deployment-specific Reliability Estimation Approach

information for our analysis. The technique that we use for recovering scenario instances and their timing information from logs is explained in Jiang et al. [21], but here we briefly summarize the technique using a sample execution log. Table III shows the first fourteen log lines from the log repository of an online bookstore. These log lines record software activities (e.g., line one), system health (e.g., line nine) as well as errors (e.g., line fourteen).

Each usage scenario in the logs consists of a sequence of steps. For example, as shown in Table III, a user registration scenario consists of the following steps:

1) A user sends a request to the web server;
2) The web server processes the request and stores the user information in a database server;
3) A confirmation email is sent out to the user.

Furthermore, each step in these scenarios shares certain identification values such as session and user ids. We need to know the frequency of different scenarios as well as how long each of them takes. Therefore, we recover the scenario instances by first abstracting log lines into execution events. Then we link related log lines (events) into sequences, whose frequency and duration can then be determined easily.

Step 1. Log Abstraction: Log lines are the output of the debug statements that developers insert into the source code. Each log line is a mixture of static and dynamic information. The static information describes the execution event (i.e., the context), whereas the varying (i.e., dynamic) parts are parameter values generated at run-time. Different values for the latter parameters cause the same execution event to result in different log lines. For example, the fourth and the twelfth log lines are generated from the same code location, but they are different since they are generated by the execution of different sessions.

In [19], we have proposed an approach that automatically abstracts log lines into execution events and marks the dynamic and static parts, such that log lines that are related to the same session can be grouped into sequences later on (step 2). Furthermore, the abstracted events can also be used to identify failure events. Table IV shows the results of log abstraction in our running example.

Step 2. Scenario Sequence Recovery: As the system handles concurrent client requests, log lines from different scenarios are intermixed with each other in the execution logs.

The log lines in Table III are generated as a result of the activities of different users: Tom, Jack and Jim. Furthermore, there are two scenarios related to Tom: user registration and catalog browsing. We recover the sequences by linking the appropriate parameter values. In our running example, we use session ids to automatically link related log lines. The results are shown in Table V. In addition, the third column of the table also shows the timestamp of the first and last steps of each recovered sequence. For example, session two started at time one and ended at time six.

TABLE V: Recovered Scenario Instances

Session	Log lines	(Start, End)	Keywords
1	1,2,5,7,10	(1,8)	Register
2	3,4,6,8	(1,6)	Browse
3	11,12	(9,10)	Browse
4	13,14	(10,11)	Update

B. State Derivation

We can now derive the system states and estimate the occurrence probability and failure probability associated with these system states based on the recovered scenario sequences from the log analysis.

We first categorize the scenario sequences into groups and identify any failures. Then, we derive the system states by taking a snapshot of the application's scenarios at a fixed time interval. The associated occurrence probability and failure probability for each state are also calculated.

Step 1. Sequence Labeling: Executing one scenario can exercise different code paths, therefore, resulting in different sequences. Hence, at the end of our Scenario Sequence Recovery step, there can be hundreds of sequences corresponding to only a handful of scenarios. We need to further reduce the amount of sequence data by properly categorizing (labeling) sequences into scenarios. We label each sequence with keywords specified by a domain expert. This process is done once for an application using keyword matching in the corresponding log entries. For new versions of an application, the domain expert might need to update some of the keyword mappings.

Our online bookstore example supports four types of operations: register, browse, purchase and update. These are the keywords used to label the sequences. When we match the keywords against each execution event in the scenario sequences, each word from the event's log entry is mapped

TABLE III: Example log lines

#	**Log Lines**
1	time=1, thread=1, session=1, receiving new user registration request
2	time=1, thread=1, session=1, inserting user information to the database
3	time=1, thread=2, session=2, user=Jack, browse catalog=novels
4	time=1, thread=2, session=2, user=Jack, sending search queries to the database
5	time=3, thread=1, session=1, user=Tom, registration completed, sending confirmation email to the user
6	time=3, thread=2, session=2, database connection error: session timeout
7	time=4, thread=1, session=1, fail to send the confirmation email, number of retry = 1
8	time=6, thread=2, session=2, user=Jack, successfully retrieved data from the database
9	time=7, thread=2, system health check
10	time=8, thread=1, session=1, registration email sent successfully to user=Tom
11	time=9, thread=2, session=3, user=Tom, browse catalog=travel
12	time=10, thread=2, session=3, user=Tom, sending search queries to the database
13	time=10, thread=3, session=4, user=Jim, updating user profile
14	time=11, thread=3, session=4, user=Jim, database error: deadlock

TABLE IV: Abstracted Execution Events and Corresponding Log Lines

ID	**Event Template**	#
E1	time=$v, thread=$v, session=$v, receiving new user registration request	1
E2	time=$v, thread=$v, session=$v, inserting user information to the database	2
E3	time=$v, thread=$v, session=$v, user=$v, browse catalog=$v	3, 11
E4	time=$v, thread=$v, session=$v, user=$v, sending search queries to the database	4, 12
E5	time=$v, thread=$v, session=$v, user=$v, registration completed, sending confirmation email to the user	5
E6	time=$v, thread=$v, session=$v, database connection error: session timeout	6
E7	time=$v, thread=$v, session=$v, fail to send the confirmation email, number of retry=$v	7
E8	time=$v, thread=$v, session=$v, user=$v, retrieving data successfully from the database	8
E9	time=$v, thread=$v, system health check	9
E10	time=$v, thread=$v, session=$v, registration email sent successfully to user=$v	10
E11	time=$v, thread=$v, session=$v, user=$v, updating user profile	13
E12	time=$v, thread=$v, session=$v, user=$v, database error: deadlock	14

into its root form (i.e. word stemming). For example, words like "browsing" and "browsed" will be mapped to the same root form "browse". The last column of Table V shows the labeled scenario names for each sequence.

Step 2. Failure Identification: We identify two types of failures in the logs based on domain knowledge. We identify and categorize failures as follows:

1) *Functional Failures (or Severity 1 failures)* are associated with system-wide outages. Examples of functional failures are system crashes, system restarts and system deadlocks. A domain expert is used to identify severity 1 failures by marking specific keywords (e.g., "thread dump") in the log files. For the example shown in Table III, there is one functional error which occurs at line 14. It is a deadlock error.
2) *Performance Failures (or Severity 2 failures)* are associated with performance slowdowns. Performance failures impact the user experience. Examples of performance failures are Service Level Agreement (SLA) violations. We basically compare the duration for each scenario's instances and see if it takes longer than the time specified by the SLA. If the SLA for the online bookstore states that all the scenarios should be executed within 5 seconds, there are two severity 2 errors at time 6 for session 1 and session 2 (as both sessions started at time 1). In [21], we presented an approach and a tool to provide automated support for rapidly identifying such failures.

TABLE VI: Derived System States and Their Failure Occurrences with Sampling Interval == 1

Time (t)	**States ($S^r(t)$) (Register, Browse, Purchase, Update)**	**Failure (-/x)**
0	(0, 0, 0, 0)	-
1	(1, 1, 0, 0)	-
2	(1, 1, 0, 0)	-
3	(1, 1, 0, 0)	-
4	(1, 1, 0, 0)	-
5	(1, 1, 0, 0)	-
6	(1, 1, 0, 0)	x
7	(1, 0, 0, 0)	x
8	(1, 0, 0, 0)	x
9	(0, 1, 0, 0)	-
10	(0, 1, 0, 1)	-
11	(0, 0, 0, 1)	x

Step 3. System States Derivation: We derive the system states by taking a snapshot at a fixed time interval based on the scenario durations. The snapshot interval must be smaller than the response time of any scenario, to ensure we do not miss any scenario information.

We define the system state from the log repository at time t $S^r(t)$ to be: $S^r(t) = (s_1, s_2, \ldots, s_n)$, where s_i is the number of active scenarios of type i at time t and n represents the total number of scenarios. The state $(0, 0, \ldots, 0)$ refers to the idle state. The state is denoted as $S^r(t)$ if it is derived from

the log repository, and as $S^a(t)$ for User Acceptance Testing logs.

In our running example, there are four types of scenarios: Register, Browse, Purchase and Update. Therefore, the system state, $S^r(t)$, is a four-dimensional state vector (s_1, s_2, s_3, s_4).

Table VI shows the derived system states from the recovered scenario instances. For example, at time 2, we have one register scenario and one browse scenario that are executing simultaneously (see Table V). Therefore, the system state at time 2 is $S^r(2) = (1, 1, 0, 0)$. In addition, Table VI also keeps track of the failure information at each time instance. As shown in the last column of this table, if there is at least one identified system failure at time t, the corresponding system state $S^r(t)$ is marked with an "x". If there is no failure at time t, the state is tagged with a "-". The states at time 6, 7 and 8 contain errors because of the SLA violation in sessions 1 and 2.

Step 4. Occurrence Probability and Failure Probability Calculations: Table VI shows the system states derived from the first 14 log lines. In practice, the log repository would contain thousands or millions of log lines. The second column of table VII shows the aggregated occurrences of all system states based on a large execution log repository.

As mentioned earlier, we need to calculate failure probabilities for the system states from the log repository and occurrence probabilities for the system states from the User Acceptance Testing. The failure probability for each state, $p_f(S_i^r)$ is calculated using the following formula:

$$p_f(S_i^r) = \frac{\#\ Failure\ Occurrences\ of\ S_i^r}{\#\ Occurrences\ of\ S_i^r} \quad (1)$$

For example, the failure probability for state $(1, 1, 0, 0)$ is calculated as: $p_f((1, 1, 0, 0)) = \frac{50}{500} = 0.1$. The 4th column of Table VII shows the failure probabilities for each system state based on the execution log repository.

To calculate the occurrence probability of a system state, $p(S_i^a)$, from the User Acceptance Testing (UAT) data, we use the following formula:

$$p(S_i^a) = \frac{\#\ Occurrences\ of\ S_i^a\ in\ the\ UAT}{Total\ \#\ occurrences\ of\ all\ states\ in\ UAT} \quad (2)$$

If we assume that the data from Table VII comes from a User Acceptance Testing log, the 2nd column of Table VII would show the number of occurrences of each system state in the User Acceptance Tests. The total number of occurrences of all the system states in the User Acceptance Tests is 4,000 (sum of entries in second column). The occurrence probability of the idle state is then calculated as $p((0, 0, 0, 0)) = \frac{2000}{4000} = 0.5$.

C. Deployment-Specific Reliability Estimation

In this section, we present our technique to provide a deployment-specific reliability estimate using Bayesian Networks. Our technique consists of the following three steps:

TABLE VIII: System States Derived from the User Acceptance Testing

System States	Occurrence Prob.
(0, 0, 0, 0)	0.5
(0, 1, 0, 1)	0.25
(1, 1, 0, 0)	0.125
(2, 3, 2, 0)	0.125

1) *System States Selection* - Identifying the system states that are common between the User Acceptance Testing logs and the log repository;
2) *Test Coverage Calculation* - Calculating the test coverage of the log repository on the User Acceptance Testing logs;
3) *Reliability Estimation* - Estimating the deployment-specific reliability using Bayesian Networks.

Step 1. System States Selection: The system states $S_i^{r=a}$ that are common between the log repository and the User Acceptance Testing are selected. Based on the occurrence probability in the real world (User Acceptance Testing) and the past failure probability (Log Repository) for these states, we can calculate the deployment-specific reliability estimates in the next step.

Table VIII shows the system states and their occurrence probabilities after analyzing data from the User Acceptance Tests. After comparing the states from the log repository (Table VII) and the User Acceptance Testing (Table VIII), there are three states in common: (0,0,0,0), (0,1,0,1) and (1,1,0,0).

Step 2. Test Coverage Calculation: For system states that are tested in the lab or in other field deployments, we know the likelihood of failure. For system states that do *not* show up in the repository, we have no prior knowledge about their failure probability. In order to obtain a lower bound estimate of the system reliability, we need to take into account for how many system states we have failure data, i.e., we need to measure test coverage.

The Test Coverage (TC) is calculated using the following formula:

$$TC(S) = \sum_{S_i^{r=a} \in S} p(S_i^{r=a}) \quad (3)$$

where $S_i^{r=a}$ are the common states between the log repository and the deployment logs; $p(S_i^{r=a})$ denotes the occurrence probability of state $S_i^{r=a}$ based on data from the User Acceptance Testing, and S is the set of covered system states. In our example, there are three states in common. Therefore, the test coverage is calculated as: $TC = 0.5 + 0.25 + 0.125 = 0.875$. The test coverage in the motivating example of Section II was 1.0, as all states that were observed in the User Acceptance Testing were also observed in the log repository.

Step 3. Reliability Estimation: Once the failure and occurrence probability for each state are calculated, we use

TABLE VII: Estimated Occurrence Probability and Failure Probability for Each System State

States S_i^r	Occurrences	Failure Occurrences	Failure Prob. $p_f(S_i^r)$
(0, 0, 0, 0)	2000	0	0
(0, 0, 0, 1)	500	100	0.2
(0, 1, 0, 0)	250	0	0
(0, 1, 0, 1)	400	0	0
(1, 0, 0, 0)	40	20	0.5
(1, 1, 0, 0)	500	50	0.1
(1, 1, 0, 1)	250	0	0
(2, 2, 2, 0)	60	0	0

Bayesian Networks to estimate the deployment-specific reliability $R(TC, S)$. Since we have no prior knowledge about the reliability of states that are not included in the log repository, we assume that all previously unseen system states derived during the User Acceptance Testing contain failures. Hence, the maximum possible reliability is $TC(S)$. This leads to a lower bound (worse case) estimate of the system reliability based on test coverage. $R(TC, S)$ is calculated as follows:

$$R(TC, S) = TC(S) - \sum_{S_i^{r=a} \in S} p(S_i^{r=a}) * p_f(S_i^{r=a}) \quad (4)$$

where $R(TC, S)$ denotes the estimated reliability given the field test coverage TC and the set of covered system states S; $p(S_i^{r=a})$ denotes the occurrence probability of state $S_i^{r=a}$ in the User Acceptance Testing; and $p_f(S_i^{r=a})$ denotes the failure probability of state $S_i^{r=a}$ based on data from the log repository.

In our running example, the deployment-specific reliability is calculated as: $0.875 - (0.5 * 0 + 0.25 * 0 + 0.125 * 0.1) = 0.8625$. Thus, the estimated deployment reliability is 0.8625. Once the reliability is estimated, deployers can match it with their targeted reliability threshold and decide whether or not to install this new version of the application.

IV. Industrial Case Study

We validate our approach by comparing our one day reliability estimate against a five day (work-week) reliability estimate that has been produced during the User Acceptance Testing of a large mission-critical application. The mission-critical application under study is a telecommunication application that is responsible for processing thousands of simultaneous client requests and has very stringent reliability requirements. This application has been deployed into hundreds of sites, which have different numbers of users, usage characteristics and reliability requirements.

For our case study, we used a repository with around 125 GB of log data. The repository contains logs derived from load and stress tests of the application and from other field deployments of the application. We also use data from the User Acceptance Testing of two different deployments, which have different configuration and usage patterns. Both deployments were tested over five days using the same version of the application. One deployment generated 4 GB of logs while the other generated 23 GB of logs. We measure the reliability of the application for both deployments using:

- five days worth of logs (full User Acceptance Testing);
- our approach, which uses just one day worth of logs and estimates the reliability using the log repository.

We then compare both reliability estimates. Due to confidentiality, we cannot list the actual estimate values. However, we note that our estimation error relative to the five-day estimate is 2.5% for the first deployment and 3.6% for the second deployment. As explained in section III, the error is an overestimate, i.e., the estimate is a safe estimate to use when deciding to deploy since the application is likely to have a higher reliability than reported by our estimate.

After examining the system state of both deployments, we note that for both deployments a small portion of the system states (5% and 8.8%) covers the majority (90%) of the occurrence probability of system states from both deployments. These small portions of system states never fail. The system states that suffer from performance slow-down are very rare, leading to a high reliability. However, if the deployed application spends considerable time in failure-prone states, we expect that our approach will provide a better estimate than the estimates based on the full acceptance testing logs. The reason for this is that the log repository contains much more accurate failure probability information than the logs from even the full acceptance testing. We draw an analogy to coin flipping to make this more clear. The more we flip a coin, the better the empirically estimated probability that a coin falls on its head side. As the log repository contains weeks or even months of system behavior, the failure probability estimated based on the log repository will be closer to the real failure probability, and therefore providing a better reliability estimate in the long run.

In summary, our deployment-specific reliability estimates using just one day of User Acceptance Testing provide a relatively good estimate with little loss in accuracy, while providing substantial savings in consulting and lab time.

V. Discussion and Future Work

In this section, we discuss the limitations of our current approach and propose directions for some future research.

A. User Acceptance Testing and Other Execution Logs

User Acceptance Testing is an important and critical step in the deployment of mission critical large-scale enterprise

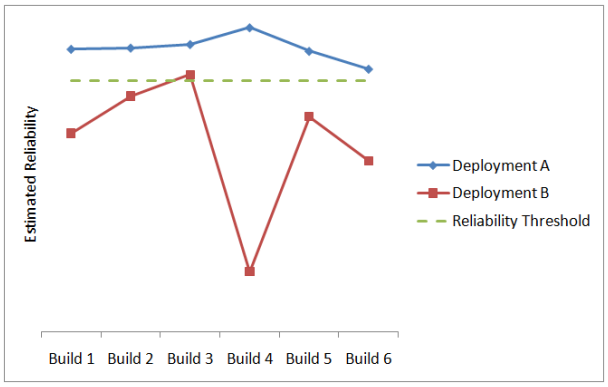

Fig. 2: Estimated Reliability for six Different Software Builds

applications. Our goal in this paper has been to reduce the amount of time spent on acceptance testing. As systems grow in size and complexity, the cost and time needed for acceptance testing grows considerably due to the need for on-site customer engagements and the need to book expensive lab time to perform such testing. Recent advances such as utility computing (e.g., Amazon EC2 [1]) help reduce some of the costs of the lab time by permitting companies to book limited time instead of having to acquire additional machines. Nevertheless, the cost of acceptance testing continues to grow.

In the future, we wish to explore the minimum number of hours that are needed to produce reasonable estimates. Using these deployment logs, we could derive the workload of the application, i.e., the occurrence probability of the system states. However, using logs from different versions of the application brings many challenges to our analysis, in particular:

1) *Performance improvement:* The new version might run faster under the same workload than the older version due to architectural and design changes. Therefore, the resulting occurrence probability distribution will shift with the application spending more or less time in different states. In this case, we cannot directly match the distribution from the old logs with the failure information from the log repository for the new version. In the future, we plan to investigate various approaches to automatically translate the old system states into the new system states.
2) *Addition of new features:* Because there is no information about the usage information in the old system for new features, we are not able to match the system states from the old logs.

B. Speeding up Pre-release Field Testing

Our approach could be used to speed up the development process by picking builds that are most suitable for limited test deployment prior to the release of an application (e.g., alpha testing). Alpha testing involves deploying the application in the field to explore its use by real users. Potential alpha candidate builds often support the main functionality, but might contain bugs in the system or miss certain features.

Figure 2 shows the deployment-specific reliabilities for two internal deployments (A and B). The dashed line in the Figure shows the minimum acceptable reliability of the alpha build of the application. For deployment A, any of the builds could be safely deployed, whereas for deployment B only the third build could be deployed safely. Using this information, the development team could start gathering user feedback much sooner by deploying the application at site A first. In addition, the team should have waited until the third build to deploy at site B and should have avoided upgrading from the currently installed version of the software in order to avoid frustrating the users due to a high chance of failure. By deploying the application earlier at site A and only at the appropriate time at site B, we are able to gain a better understanding of the reliability and usage characteristics of the application in a real-life setting. This information would help improve future builds and speed up the development process.

To conduct our analysis across builds, we must ensure that the performance between builds does not fluctuate considerably. We used our previous work [21] to verify that the performance between the consecutive builds was consistent (no major shifts). Then we used logs generated from the load and stress tests of each build and logs from each deployment site to customize the reliability estimates.

VI. Threats to Validity

This section discusses various possible threats to the validity of our approach.

A. Construct Validity

1) Snapshot Interval: To accurately capture the change of system states, we need to pick a snapshot interval shorter than the shortest response time of any scenario in our application. Shorter intervals lead to more accurate system states. However, the length of the snapshot interval is limited by the logging interval. In our industrial case study, the timestamp in the execution logs is accurate up to milliseconds. However, if we pick the snapshot interval to be every millisecond or every 10 milliseconds, the state derivation step will lead to a huge number of repeated states with no changes between them. For this reason, we picked a one-second interval. Our state derivation step and our analysis finish within one hour on a Quad-core machine. We cannot pick any interval longer than 1 second, as most of the scenarios finish within 1 to 2 seconds.

2) System States: As there can be an infinite number of combinations of different workload requests, there can be an infinite number of system states. However, in our large industrial study, there are only a limited number of system states. Furthermore, the distribution of system states follows the Pareto-principle: a small percentage (5% and 8.8%) of states covers the majority (90%) of the system behavior. However, many of the states in the remaining 10% of the system behavior can be equivalent, so we plan in the future to apply fuzzy-clustering techniques on these states to group them and further reduce the number of system states.

In this paper, we only consider the failure probability and occurrence probability of system states, since we believe that

a system fails or slows down due to the current state (i.e., heavy workload). In the future, we plan to look into transitions between system states to see whether they might be the reason for system failure. In particular, a state might be a failure state just because of the previous states that eventually lead it to fail.

B. Internal Validity

1) Contingency of Workload: Our deployment-specific reliability estimates are based on the workload in the field deployment at a particular point in time. In reality, this workload might shift over time leading to different reliability estimates. Our approach could be used to track such shifts in workload and warn about the impact of such a shift on the reliability of the application.

One method of checking if the application's behavior changes over time, is to check the distribution of system states over time. If the distribution of system states changes over time, then the system reliability will likely change as well. The distribution of system states remained stable in both of our case studies (the alpha testing and the User Acceptance Testing in two different sites).

2) Limitation of Execution Logs: In the studied mission-critical application, there is a dedicated component that monitors the overall health. Thus, business level information as well as error messages are logged. Our current approach may not work if the studied application does not log all the necessary system information. In addition, we assume that all the errors are recorded in the logs, which is usually true for large mission-critical applications. Last but not the last, not all errors reported in the logs are operational or performance-related. Errors from which the application recovers are not considered as failures. Examples of such errors are temporary communication failures that do not impact the overall system health or SLA. In this paper, we manually verified all failures included in our analysis.

C. External Validity

In this paper, we introduced a novel approach to reduce the time and cost needed for User Acceptance Testing. We evaluate our approach against a large mission-critical telecommunication application and showed that our estimate lies within 4% of the more traditional longer User Acceptance testing. To show the general applicability of our approach, we should evaluate it on other large mission-critical applications. However, such applications are usually developed by large commercial companies and their data is hard to obtain due to legal and confidentiality concerns.

VII. Related Work

In this section, we discuss two areas of work related to our log-based empirical reliability estimation approach: approaches that estimate the system quality in the field and approaches that use log analysis.

Empirical Estimation of Software Availability and Reliability

Mockus [25] uses information from operational customer support systems to estimate the availability of a large telecommunication system. Information from customer support systems is more straight-forward to obtain than extracting failure information from the execution logs. However, the customer support systems may not contain all failure information, as they miss externally unnoticeable problems and problems that are not reported by customers. Nagappan et al. [17], [27] provide a reliability estimate based on information from the static source code metrics and dynamic test coverage. Their approach is implemented as an Eclipse IDE plugin to provide rapid feedback for unit testing.

Avritzer et al. [8], [10], [11] have proposed several approaches for generating test suites based on the operational profile and for estimating the reliability of mission-critical applications. In [10], Avritzer et al. use Markov chains to generate load testing suites based on an operational profile. In [8], [11], Avritzer et al. introduce a transient analysis of the failure-based Markov chain to model reliability decay as a function of time. Our approach is similar to [8], [11], as we incorporate the occurrence probability distribution from the deployment and the failure probability from the repository. However, rather than using the failure information from vendors, we use the log framework and a large data set of field data to empirically derive failure and occurrence probability distributions. In addition, since we have the actual logs to estimate the occurrence probability of system states, we do not need Markov chains to model workload usage.

Log Analysis

In general, there are two sources of non-invasive data that can be used to understand, monitor, and analyze the various aspects of the system behavior: execution logs and performance logs.

Execution logs are generated by output statements that developers insert into the source code. Execution logs are widely available and are often used for remote issue resolution and for legal compliance purposes (e.g., "Sarbanes-Oxley Act of 2002" [3]). Aguilera et al. [4], [28] developed various algorithms to perform black-box performance debugging on distributed systems. They use the header information of the TCP packet traces (source, destination and time) to infer the dominant causal paths through a distributed application. Unfortunately, the accuracy of the inferred causal paths decreases as the degree of parallelism increases. Marwede et al. [23] use timing anomalies to automatically uncover functional problems. Hassan et al. [18] propose a light-weight approach to extract customer operational profiles from the execution logs. Jiang et al. [20], [21] analyze execution logs to detect functional and performance problems in the load tests. The log analysis approach presented in this paper is related to the approach presented in [9], because in both papers we abstract the log information into a set of different system states. However, in [9] the state definition used was the set of rules that were fired as a result of a change in object memory, while in this paper the system state is defined as a set of active scenarios present in the application at a particular moment in time.

Performance logs, which are generated by third party monitoring tools like PerfMon [2], record the system resource

utilizations like CPU, memory and disk. Avritzer et al. [7], [6] propose algorithms to detect the need for software rejuvenation by monitoring the changing values of performance metrics. Mi et al. [14], [24] and Cohen et al. [15], [31] develop application signatures based on various system metrics (like CPU and memory usage). The application signatures are further used for efficient capacity planning and anomaly detection. The main difference between these approaches and ours is that we use execution logs for our analysis. Execution logs provide more in-depth domain-specific information.

VIII. Conclusion

Studies show that many field failures are due to load problems rather than functional problems. One system can be deployed in hundreds or thousands of customer sites, each with a different workload. As it would be impossible to test all the possible workloads in the lab, User Acceptance Testing is becoming an important step in the deployment of large mission-critical applications. However, User Acceptance Testing is costly and time-consuming, as it involves costly on-site customer engagements while running lengthy (multi-day) tests using precious customer lab resources.

In this paper, we propose an approach that reduces the User Acceptance Testing time, while providing a relatively accurate reliability estimate, based on mining the readily available large sets of execution logs. Experiments on a large mission-critical application show that our reliability estimates are within a 4% error of the estimates produced by longer-running full Acceptance Testing processes.

Acknowledgement

The authors thank Dr. Bram Adams for his useful feedback on this paper.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] PerfMon Sample. http://msdn.microsoft.com/en-us/library/aa645516(VS.71).aspx.

[3] Sarbanes-Oxley Act of 2002. http://www.soxlaw.com/.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[5] M. Audris, P. Zhang, and P. L. Li. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[6] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyuker. Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[7] A. Avritzer, A. Bondi, and E. J. Weyuker. Ensuring stable performance for systems that degrade. In *Proceedings of the 5th international workshop on Software and performance*, 2005.

[8] A. Avritzer, F. P. Duarte, R. M. M. Le ao, E. de Souza e Silva, M. Cohen, and D. Costello. Reliability estimation for large distributed software systems. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research*, 2008.

[9] A. Avritzer, J. P. Ros, and E. J. Weyuker. Reliability testing of rule-based systems. *IEEE Software*, 13(5), 1996.

[10] A. Avritzer and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9), Sep 1995.

[11] A. Avritzer and E. J. Weyuker. The automated generation of test cases using an extended domain based reliability model. In *Proceedings of the ICSE Workshop on Automation of Software Test, 2009 (AST 2009)*, 2009.

[12] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[13] L. Bertolotti and M. C. Calzarossa. Models of mail server workloads. *Performance Evaluation*, 46(2-3), 2001.

[14] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2008.

[15] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 2005.

[16] Compuware. Applied Performance Management Survey, Oct 2007.

[17] M. Davidsson, J. Zheng, N. Nagappan, L. Williams, and M. Vouk. Gert: An empirical reliability estimation and testing feedback tool. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.

[18] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz. An industrial case study of customizing operational profiles using log compression. In *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[19] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal on Software Maintenance and Evolution: Research and Practice*, 20(4), 2008.

[20] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, 2008.

[21] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, 2009.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[23] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, 2009.

[24] N. Mi, L. Cherkasova, K. M. Ozonat, J. Symons, and E. Smirni. Analysis of application performance and its change via representative application signatures. In *Proceedings of the Network Operations and Management Symposium*, 2008.

[25] A. Mockus. Empirical estimates of software availability of deployed systems. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 2006.

[26] J. D. Musa, A. Iannino, and K. Okumoto. *Software reliability: Measurement, prediction, application*. McGraw-Hill, 1987.

[27] N. Nagappan, L. Williams, and M. Vouk. "good enough" software reliability estimation plug-in for eclipse. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, 2003.

[28] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th International Conference on World Wide Web*, 2006.

[29] J. Voas. Will the real operational profile please stand up? *IEEE Software*, 17(2), 2000.

[30] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12), 2000.

[31] S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.