A RANDOM-WALK SIMULATOR FOR RELIABILITY MODELING TRAINING

EDWARD L. JONES
Department of Computer and Information Sciences
Florida A&M University
P.O. Box 164
Tallahassee, FL 32307 USA

AUBREY J. REMBERT
Department of Computer and Information Sciences
Florida A&M University
P.O. Box 164
Tallahassee, FL 32307 USA

ABSTRACT
Software reliability is the probability that a system performs its purpose adequately for a specified period of time under specific operating conditions. This paper presents RANDREL, a Java-based simulator that provides an environment for training students in reliability testing. Reliability testing is modeled as a random walk through the program execution state space into which faults have been seeded at specific points. When a fault is encountered during the simulation, a failure is reported, the tester fixes the fault and resumes reliability testing. After each failure the reliability growth model is updated and used to determine whether the reliability goal has been achieved. This paper describes the design of RANDREL and early experiences using it.

KEYWORDS
Reliability modeling, simulation, training, testing.

1 INTRODUCTION
Reliability of software is defined as the probability of a system performing adequately for the period of time intended under specific operating conditions [1, 2]. The combination of the time period and the operating conditions represent the mission for which reliability testing is performed. Adequate system performance is defined by the system specification. Finally, the operating conditions also include environmental elements such as machine and human interfaces, skill level of the system users, and the number and mix of users.

1.1 Reliability Measurement
There exist many models for expressing the reliability of a system in quantitative terms. Software reliability is one of the few "-ilities" of software quality that can be measured. The common, fault-based measure of quality such as faults/KLOC is of limited utility in reliability measurement [3]. A software fault is defined as an intrinsic error in the software [1] which, when encountered, leads to failure. A software failure is defined as deviation from the specification. A fault in an instruction has a bearing on mission reliability only if that instruction is required to perform the mission.
A measure like mean time between failures (MTBF), which relates more directly to a specific mission, is a better measure of reliability. Cumulative MTBF, the average time interval between failures taken over all failures since the start of reliability testing, provides meaningful information about the growth of reliability in a system. Reliability growth can be plotted using packages such as the Duane Plot Model formula [5], and used to project the time at which a specific MTBF value will be attained [2].

1.2 Reliability Testing
Preparation for reliability testing requires a careful definition of the mission, and the definition of test scenarios that represent system utilization for the mission. A clear statement of what constitutes a software failure during the mission must be agreed upon prior to testing. Techniques such as operational profiles are used to model system usage patterns associated with the mission. These activities require significant up-front cost.

The basic steps performed during reliability testing are:

1. Test execution – executing the system under the workload typical of the mission.
2. Failure detection – inspecting test results to identify system responses that do not meet the specifications.
3. Fault discovery – finding the fault associated with failures.
4. Fault removal – correcting the fault(s) in the code that caused the failure.
5. Reliability assessment – updating the reliability growth model and determining whether reliability goals have been reached.

This may seem like a very simple process, but it requires considerable time and effort to perform these tasks. Experiencing a failure during reliability testing triggers a chain of events. The offending piece of code must be
isolated and candidate faults hypothesized. This step may take minutes, hours, or, in complex system, even days. The candidate faults must be reduced to one or more that are most likely to cause the observed failure. The possible corrections of these faults are evaluated, and the best corrections are implemented. The diagnosis of the failure and the attempted correction must be documented. New test cases must be designed for the code change, and the modified code must be tested. Finally, regression testing may be required to ensure the correction did not introduce additional faults. This sequence of activities shows how involved reliability testing can become.

2 SIMULATION BASED RELIABILITY TEST TRAINING

In order for students to learn reliability testing, they must have the correct motivation and tools to gain an appreciation for the practice and conceptual aspects of reliability testing. Students should be afforded an opportunity to learn about software reliability testing, but since reliability testing takes so much time and effort, rarely do students gain practical knowledge of the subject while in school. Instead, they are usually taught “on the job,” often in pressure situations. If effective tools were developed to provide students with some practical experience of reliability testing, they would be better prepared for situations that will arise on the job.

Aircraft pilots learn how to fly in flight simulators. This type of training avoids risks or expenses involved with flying for real. The simulation does not last the actual length of the flight, but trainees are exposed to and become familiar with certain types of problems before they actually have to fly. The simulator can model the actual system except for the pilot. The prospective pilot can interact with the simulator as if they were actually flying, experiencing all the difficulties without any of the risk [4].

This same approach can be applied to teaching students the key aspects of reliability testing. They can experience the challenges of reliability testing, in a reduced period of time. They also can experience situations in which the reliability goal is not attained. Simulation can aid students in learning fundamental reliability testing concepts.

2.1 RANDREL Simulator System

The objectives of RANDREL are to give students practical exposure to reliability testing through simulation, to encourage students to engage in reliability testing activities, and to increase the awareness of importance of reliability testing. Cumulative MTBF is the reliability measure used by RANDREL. This measure presents the most relevant information to a novice reliability tester.

The reliability tester controls the simulation through a graphical user interface by selecting appropriate parameters for the test. As shown in Figure 1, the parameters are:

- **Lines of code to be tested.** The lines-of-code parameter in RANDREL ranges from 1000 to 60000 inclusively.
- **Fault intensity of the software.** This parameter ranges from 1% - 5% inclusively. The fault intensity parameter in an actual reliability test would come from the analysis of the fault intensity data gathered during unit testing or from historical project data.
- **Mission time of the software.** This parameter specifies the minimum error-free running time to complete the mission the software must accomplish. Time is represented by a fictitious measurement called TU, which is related to a step taken in the random-walk simulation.
- **Target cumulative MTBF.** This parameter specifies the reliability goal of the overall reliability test.
- **Minimum number of tests.** This parameter specifies the minimum number of test iterations, to ensure that a large enough sample is examined to make the results acceptable.
- **Time constraints of the test.** This parameter specifies the maximum time available for reliability testing. In an actual reliability test, there exists a schedule constraint limiting the time for reliability testing.

2.2 RANDREL Simulation

The RANDREL simulator represents the program execution state space as a 2-dimesional square grid. The length of the grid is proportional to the square root of the number of lines of code. Once the program state space grid is created, faults are randomly seeded at x-y positions. Three faulty severity levels are recognized:

- **Data error** – the fault leads to incorrect data values
- **Loss of state** – the fault causes the software to lose state information
- **Fatal** – the fault causes abnormal termination.

RANDREL simulates program execution by initiating a random walk at the origin. Each step in the walk is a unit step, changing one coordinate by ±1. There is no target position that terminates the walk. Instead, the walk terminates upon making a specified number of steps, or upon encountering a fault with a specified severity level. When the simulation lands on a fault location, RANDREL records the failure in the failure log, along with information about the location of the fault.
2.3 RANDREL Design Patterns

The implementation of RANDREL contains design patterns and uses refactoring techniques [6]. The design separates the user interface, system control, and system information in the Model-View-Controller pattern (MVC). By separating the components of RANDREL in this fashion allows flexibility, and enables RANDREL to evolve. The simulation and computational model, RandRelModel, contains all of the key data and performs all necessary data manipulations. The RANDREL user interface, RandRelView, displays the data that is contained in RandRelModel. This component does not perform any data manipulation. The controller, RandRelController, handles all of the user-initiated actions and delegates all data manipulation to RandRelModel.

RANDREL also uses the factory pattern when deciding what type of fault to seed into the program – data fault, loss-of-state fault, or fatal fault. The FaultType factory returns a random fault type based on preset probabilities. The behavior of that fault during debugging (fault seeding [1]) and removal is invoked polymorphically. The use of design patterns and refactoring make RANDREL both malleable and understandable.

2.4 RANDREL SIMULATION EXAMPLE

The simulation begins with the tester entering parameters for the software under test and the reliability testing constraints as shown in Figure 1. Once the parameters are entered, the tester must Commit the parameters (make the parameters immutable while testing is being conducted). The tester then proceeds to the Simulate button that starts the simulation. When the simulation begins, the tester will receive a message that indicates that the mission was completed or that there was a failure during.

An attempt to complete the mission is called a test run. A simulation trace is captured which contains the sequence of random-walk positions traversed during the test run. After each test run, reliability information about the software under test is recomputed based on the current test run, along with results from all the previous test runs.
After the end of each test run the time remaining to test is updated and the number of failures encountered is recorded. If a failure occurs, the tester must proceed to the failure log to view the information about the failure. In an actual reliability test this information would be manually recorded, but to achieve the desired goal of time saving, RANDREL logs the appropriate information. The information logged about the failure includes where the failure happened, the sequence number of the failure, the time of the failure according to the system age not the test run age, and the cumulative MTBF at the time of occurrence. The tester then examines the simulation log to identify the location of the fault associated with that failure. The tester attempts to fix the fault by removing it from the state space of the random walk. The tester may observe the reliability growth model (cumulative MTBF) plot, which has been updated to reflect the history of failures throughout the period of reliability testing. Testing is terminated when the target cumulative MTBF has been reached, or when the limit on total testing time has been reached. The overall testing is judged successful only if the reliability goal has been reached.

Figure 2 shows the result of a simple experiment using RANDREL. The simulation took approximately ten minutes to complete. The parameters of the simulation are given below:

- Lines of code = 1000.
- Mission time = 1000 time units.
- Fault-density = 2%.
- Target MTBF = 800 time units.
- Testing time constraints = 5000 time units.
- Minimum number of tests = 5.

The final cumulative MTBF was 303 time units. The number of test runs performed was 14. The number of
failures found was 13. The number of failures remaining was 9. The growth model of the simulation is shown in Figure 2.

3 CONCLUSION AND FUTURE WORK

The goal of RANDREL is to reduce the time it takes to teach and experience reliability testing. It began as a semester project for a graduate course in verification and validation. The initial, text-based prototype for RANDREL was built in C++. From this prototype, it was determined that Java would be a much better choice in terms of interface, ease of organization, and lines of code. RANDREL contains 22 java classes that aid simulation and "bebugging" (fault seeding). The design of RANDREL is expected to evolve to exploit future uses of the simulation technique. More research needs to be done in the area of reliability testing and simulation to improve the model. Experimental data must be examined more closely to find out what is actually being learned during the simulation.

On-going and future work with RANDREL includes following activities, and extensions.

- Validate the model for realism by studying the effects of fault density and program size on reliability growth and testing costs.
- Model operational profiles, which restrict access to portions of a software product, by a guided random walk with regional "walk rules" that bias the direction of a walk.
- Visual depiction of program state-space coverage achieved during reliability testing, to study the effects of operational profiles.
- Provide user-defined fault severity levels.
- Implement training controls that make the simulation more realistic by letting tester discover faults by looking at the execution trace data.

4 ACKNOWLEDGMENT

The authors acknowledge the encouragement and feedback from students in the graduate Verification and Validation course. This work was partially supported by National Science Foundation grant EIA-9906590, and by corporate grants from Dell Computer, and Lucent Technologies.

REFERENCES