A Tutorial on Software Reliability Engineering

CMG '89

J. Dennis Lawrence
Lawrence Livermore National Laboratory

ABSTRACT[1]

Computer systems are increasingly being required to operate free from externally-detected failures. This is particularly true in the case of large distributed real-time and transaction-processing systems in application areas such as transportation, medicine, defense, finance, marketing and communications. There are many similarities in approaches and techniques between performance analysis and reliability analysis, so the local Performance Expert is a natural person to approach for help with reliability problems.

This tutorial will provide an introduction to Reliability Engineering, primarily as it applies to software development. It consists of four parts. The first part defines some important terms, and describes some ways to classify faults and failures. The second portion describes various kinds of activities that can be carried out during the life cycle of the application system, especially those that will be of interest to the Reliability Expert. The fourth section discusses various techniques for achieving software fault tolerance. Finally, we discuss a few modeling techniques and the problems of accurate data collection.

1. INTRODUCTION

Since the first computer program was written, the programming activity has been dominated by the problem of constructing reliable software that satisfies the user's requirements while meeting cost and schedule constraints. In recent years, and into the foreseeable future, this emphasis on reliability will become even more important to our profession. There are many reasons for this increasing importance [1]. (1) The volume of computer systems being developed is growing at an increasing pace. (2) Applications are becoming ever more visible to the general public – that is, your customers – as interactive systems proliferate and computer control enters the most common everyday appliances. (3) The complexity of applications is growing at an increasing rate. (4) In many areas, the risks to life, health, property and wealth when software fails is increasing. Many new applications involve several of these areas at the same time, compounding the problem. International banking, air traffic control and many military systems are examples of this.

There are interesting analogies between engineering software to meet reliability objectives and engineering software to meet performance objectives. The same general types of activities are involved, and the same sorts of skills are required. In both cases, the analyst should work with the software development team throughout the project life cycle. Measurement data must be collected, evaluated and analyzed. Models must be constructed

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

and validated. There are similar problems relating to project costs, schedules and manpower. In both cases, education of management, users and computer professionals is required.

On a detailed level, there are (of course) many differences. These similarities and differences are explored in this tutorial.

To begin with [2, 3], the reliability of a computer system is determined by several different phases. First, we try to keep faults out of the system. Second, since some faults will escape this effort, we try to identify and eliminate them. Third, since neither design nor test is perfect, and since some new faults will occur during operation due to operational or environmental faults, we attempt to cope with them once they appear. There is a useful analogy here to the security of (say) a bank. We try to keep most robbers out; stop those that get in the door; and recover the loot from those that get away. We are also forced to recognize that there are some successful robbers, who escape to enjoy the proceeds.

Fault Avoidance is concerned with keeping faults out of the system in the first place. It will involve selecting techniques and technologies which will help eliminate faults during the analysis, design and implementation of a system. This includes such activities as the selection of high quality reliable hardware and the use of formal conservative system design tactics. Good management practices are essential to achieving fault avoidance.

Fault Removal techniques are necessary to eliminate any faults which have survived the fault avoidance phase. Testing is the usual technique. Testing principles and procedures are well known, and will receive little elaboration here.

Fault Tolerance is the last line of defense. The intent is to incorporate techniques that permit the system to detect faults and avoid failures, or to detect faults and recover from the resulting errors, or at least warn the user that errors exist in the system.

Graceful Failure. In spite of our best efforts, computer systems will fail. When this happens, we want to keep the cost acceptable, and restore the system to operational status within a pre–specified period of time. Of course, what cost is "acceptable" is situation-dependent. A routine batch reporting run can accept a much more abrupt termination than a railway control system.

2. TERMINOLOGY

2.1. Some Basic Definitions

2.1.1. Definitions

The failure of a computer system involves three separate events, which may be widely separated in time. First, something basic goes wrong; a wire breaks, or a bug is coded into a program. At some later time (which may vary, depending on the cause of the problem, from microseconds to years), this results in an incorrect internal state of the application system. At an even later time, the problem becomes visible to the user of the system (which may be another program, a mechanical or electronic device, or a human being). There are three words that are generally used for these events: fault, error and failure. Unfortunately, there is no agreement on which word should be used for which type of event. For this tutorial, I will use the following definitions [3, 4, 5].

A *fault* is a deviation of the behavior of a system from the authoritative specification of its behavior. A *hardware fault* is a physical change in hardware that causes the system to change its behavior in an undesirable way. A *software fault* is a mistake (bug) in the code. A *procedural* fault consists of a mistake by a person in carrying out some procedure. An *environmental fault* is a deviation from expected behavior of the world outside the computer system; electric power interruption is an example.

An *error* is an incorrect state of hardware, software or data resulting from failures of components, software bug, physical interference from the environment, operator mistake, or incorrect design. An error is, therefore, that part of the system state which is liable to lead to failure. Upon occurrence, a fault creates a *latent error*, which becomes *effective* when it is activated, leading to a failure.

A *failure* is the external manifestation of an error within a program or data structure. That is, a failure is the external effect of the error, as seen by a (human or physical device) user, or by another program.

2.1.2. Reliability Measures

It is best, of course, if measures can be given in quantifiable terms. We shall use the following [4, 5]:

The *reliability*, $R(t)$, of a device or system is the conditional probability that the device or system has survived the interval [0, t], given that it was operating at time 0. Reliability is often given in terms of the *failure rate*, $\lambda(t) = -\dfrac{d \ln R(t)}{dt}$, or the *mean time to failure*, $mttf = \displaystyle\int_0^\infty R(t)\, dt$.

Two terms in the above definition deserve some comment. In practice, it is not always easy to get users to agree on what constitutes a system failure. However, in order to calculate reliability over time, a consistent definition is necessary. What that definition is probably matters less than the act of choosing one and sticking to it. Survival, then, is the absence of failure, and may (if you so choose) include periods of degraded service. Since reliability is a function of time, a careful choice of time periods must also be made and not changed. Possibilities include clock (elapsed) time, CPU time and instruction count.

The *availability*, $A(t)$, of a device or system is the probability that the device or system is operational at the instant of time t. For nonrepairable systems, availability and reliability are equal. For repairable systems, they are not.

The *maintainability*, $M(t)$, of a device or system is the conditional probability that the device or system will be restored to operational effectiveness by time t, given that it was not functioning at time 0. Maintainability is often given in terms of the *repair rate*, $\mu(t) = -\dfrac{d \ln (1 - M(t))}{dt}$.

The *safety*, $S(t)$, of a device or system is the conditional probability that the device or system has not encountered a catastrophic failure by time t, given that there was no catastrophic failure at time 0. Sometimes safety and reliability are in conflict: increasing one may result in decreasing the other. This is not considered further here.

2.2. Some Types of Faults

2.2.1. The Persistence of Faults

Any fault falls into one of the following three classes [6]:

- A *design fault* is a fault that can be corrected only by redesign. Most software faults are design faults.

- An *operational fault* is a fault where some portion of the system breaks during the operation of a device or system. The breakage must be repaired in order to return the system to an operational state.

- A *transient fault* is a fault that does cause a system error, but is no longer present when the system is restarted. An example is communication line noise. In computer systems, up to half of all failures are transient, or appear transient.

A system is constructed according to some specification. If the system fails, but still meets the specification, then the specification was wrong. This is a design fault. If, however, the system ceases to meet the specification and fails, then the underlying fault is an operational fault. A broken wire is an example. If the specification is correct, but the system fails momentarily and then recovers on its own, the fault is transient.

2.2.2. The Source of Faults

Sources can be classified into a number of classes; ten are given here. For each one, we describe the source briefly, and mention which types of persistence are possible.

- A *hardware fault* is a fault in a hardware component, and can be of any of the three persistence types. For application systems, however, we rarely encounter hardware design faults. Transient hardware faults are very frequent.

- A *software fault* is a bug in a program. In theory, all such are design faults.

- An *input data fault* is a mistake in the input. It could be a design fault (connecting a sensor to the wrong device is an example) or an operational fault (if a user supplies the wrong data).

- A *permanent state fault* is a fault in state data that is recorded on non-volatile storage media (such as disk). Both design and operational faults are possible.

- A *temporary state fault* is a fault in state data that is recorded on volatile media (such as main memory). Both design and operational faults are possible.

- A *topological fault* is a fault caused by a mistake in system structure, not with the component parts. All such faults are design faults.

- An *operator fault* is a mistake by the operator. Any of the three types are possible.

- A *user fault* differs from an operator fault only because of the different type of person involved; operators and users can be expected to have different fault rates and distributions.

- An *environmental fault* is a fault that occurs outside the boundary of the system, but that affects the system. Any of the three types is possible.

- An *unknown fault* is any fault whose source class is never identified. Unfortunately, many faults occur whose source cannot be identified.

2.3. Some Types of Failures

2.3.1. The Scope of Failures

Failures can be assigned to one of three classes, depending on the scope of their effects [7]:

- A failure is *internal* if it can be adequately handled by the device or process in which the failure is detected.

- A failure is *limited* if it cannot be so handled, but if the effects are limited to that device or process.

- A failure is *pervasive* if it results in failures of other devices or processes.

2.3.2. Failure Modes

Different failure modes can have different effects on a system. System specification of performance affects the way in which component failure modes will affect the system. The following definitions apply [5].

- *Sudden failure* – failure that could not be anticipated by prior examination.

- *Gradual failure* – failure that could be anticipated by prior examination.

- *Partial failure* – failure resulting in deviations in characteristics beyond specified limits but not such as to cause complete lack of the required function.

- *Complete failure* – failure resulting in deviations in characteristics beyond specified limits such as to cause complete lack of the required function. The limits referred to in this category are special limits specified for this purpose.

- *Catastrophic failure* – failure which is both sudden and complete.

- *Degradation failure* – failure which is both gradual and partial.

3. LIFE CYCLE ACTIVITIES

System reliability is a very broad concept, and includes many aspects that are beyond the scope of this tutorial. Such activities as software engineering, testing procedures, management activities and the like are, of course, very important to achieving reliable computer systems. Some of these are mentioned below for completeness; however, the emphasis is on activities that specifically address reliability.

The various life cycle stages can be treated within the paradigm of fault avoidance – fault removal – fault tolerance – graceful failure. For example, one can look at the Requirements Specification Phase of the life cycle as an attempt to avoid faults during later phases. A formal review of the requirements document is intended to remove faults from that document. An iterated life cycle, whereby one cycles back to earlier phases, is a form of fault tolerance: when errors in the requirements specification are discovered in later phases, a formal process will exist for correcting them, and correcting any consequences in design, implementation and so on.

3.1. Requirements Specification Phase

Software reliability engineering must begin at the very beginning of the life cycle. The actions and activities suggested here are not meant to be exhaustive, and generally omit well-known ideas (such as, "requirements must be testable"). Notice how similar these suggestions are to those made for software performance engineering.

- Identify imposed restrictions on boundaries among hardware, software, humans, and the environment. That is, what is inside the system, what is outside it, and what information is transmitted across the boundary. For reliability considerations, pay particular attention to unexpected input, abnormal operating conditions, and other strange environmental effects.

- Determine which portions of the computer system are repairable and which are not. Either of these classes may be empty, of course. Additional care will be required to achieve high levels of reliability in the nonrepairable portions of the system. An on–board computer in a space craft is an example; is there any mechanism for correcting software faults found during the mission?

- Quantify all reliability, availability and maintainability goals. This includes goals for hardware, software, users, and the system as a whole. The software cannot be considered in isolation. System behavior in these respects depends on the behavior of the components and on the topology of the system. Goals should be realistic, since there comes a point at which further improvement in reliability requires a disproportionate increase in cost.

- Quantify fault tolerance goals. What types of system failures must be accepted, and dealt with? What data can we not afford to lose? How much time may recovery be allowed to take?

- Identify the means of achieving fault tolerance. Fault tolerance is achieved by some form of redundancy. This, of course, increases the cost of the system; some degree of balance is generally required. Redundancy can occur in hardware, software, information (data), operations, and time. In general, a mixture is necessary. Strictly speaking, this is a design activity. However, many of the issues will involve system users and customers; these can best be addressed as part of the requirements.

- Identify operational sequences that system users are expected to carry out. Analyze each for reliability. This is a human factors problem, and has many ramifications. In particular, how should the system respond to incorrect sequences of commands?

- Identify each timing constraint. In each case, decide how the system will know if the constraint has been violated. What should be done if the timing constraint has not been met?

- Determine the implications of performance goals on fault tolerance. One way to achieve fault tolerance is to use redundant hardware. Upon partial system failure, degraded operation may result. Can the performance goals be met in such situations? If not, this may result in additional failures, causing a cascade into total failure. Careful analysis is required.

3.2. Software Design Phase

Software reliability should be designed into the system. Any attempt to retrofit reliability into a system is a very costly and imperfect procedure. One aspect of any design, of course, is to ensure that each requirement can be met. Reliability requirements are no different from any others in this respect. When the design is complete, you should be able to demonstrate how each requirement is satisfied by the design.

- Analyze each requirement for reliability. What types of faults can cause the requirement not to be met? What will be the scope of any failure? What kind of failure modes may occur, and how can the less desirable modes be avoided?

- When drawing information models (such as entity relationship diagrams [8]), consider what redundant information should be included in order to help achieve recovery and performance goals.

- Many authors recommend that data flow diagrams (DFD) not include error paths. (See [9], for example.) This is a bad idea when dealing with systems with high reliability requirements; error and recovery paths may well end up as the major portion of the diagram. This must be considered as part of design. There are two basic reasons for this. First, error detection and handling may well occupy the larger portion of the DFD. Second, it may well be necessary to change the order of processing to achieve fault tolerance.

  Similar comments apply to control flow diagrams and state transition diagrams.

- If checkpoint, logging and recovery techniques are to be used, the details must be carefully worked out. What part of the system state will be saved during the checkpoint? Where – on the same system or a remote system? Where will transaction logs be placed? Careful analysis has been done regarding transaction commitment in the context of database systems; much of this will apply to application systems. In particular, choosing the checkpoint frequency can be complex [10].

- All input should be checked for validity, to the fullest extent possible. This cannot be done, of course, unless it is known how to determine if the input is valid. This information may be hard to acquire.

- When performing any sort of communication (process-to-process, CPU to disk, process-to-terminal, ...), the program will need to know if the recipient fails to receive the message. This involves some form of (positive or negative) acknowledgement from the recipient, as well as time-outs to detect cases when the recipient cannot be reached.

- How will reliability be measured during test and operation? What will be measured? This needs to be worked out at the design phase so that the measurement process can be planned out in advance of need.

- All failures detected by the system should be recorded in a failure log, for later analysis. That is, the code should be thoroughly instrumented. The capture, recording and analysis of failure records needs to be thought out. For example, the log should be on a CPU other than the one containing the program in question – how can this be done? How will failures not detected by the program (such as power failure) be entered into the log?

- Each element of the system design needs to be analyzed for reliability. The lists given in Sections 2.2 and 2.3 can be helpful. Where can faults arise that may affect this design element? How persistent can they be expected to be? Recovering from a transient communication fault will normally be quite different from recovering from a topological design fault. How will the program cope with these differences? What sorts of faults will be ignored, on the grounds that the expense of handling them cannot be justified by the frequency and severity of their occurrences? How can the scope of failures caused by a particular fault be limited?

- It may well be advisable to create a formal reliability model of the system. Modeling techniques include reliability block diagrams, fault trees, Markov models, Petri net models, reliability growth models, and others. This is discussed later in the tutorial.

- System reliability can frequently be increased by the use of redundant I/O devices and redundant communication paths. The cost of this redundancy will have to be balanced against the cost of system failure. The latter is equal to the actual cost of a failure multiplied by the failure rate and the operational lifetime.

- If the application system controls devices over a wide geographic area, measures should be taken to prevent failures in one portion of the system from propagating to other areas. The term "wide" is system-dependent in this context. It may be possible to logically or physically isolate the area that is in trouble, and continue to control other areas. At worst, this should permit graceful shut-down of the other areas. The principle is: a local failure should not cause a total system failure.

- Some portions of software are usually more important than others. Failure of a less important task should not impact the existence of more important tasks (although it may influence what they do – recover from the failure, for example).

- It may be possible to have a monitor task, whose purpose is to periodically send messages to all other tasks. These will look for such messages from time to time. When one arrives, the recipient will return a message verifying its status. If no reply is received by the monitor after a period of time, the recipient is deemed dead (or in a loop); it is killed and restarted. This may have implications in other portions of the system, as well. The monitor may reside on a remote node, and may be duplexed on several remote nodes, for redundancy.

- If a portion of the computer control becomes impossible because of conditions in the environment, the affected areas and functions should be detached from system control, so that other areas and functions can continue to operate. Once these conditions are corrected, computer control should be resumed in a "smooth" manner; this last word obviously needs to be quantified.

3.3. Implementation Phase

During the coding phase, scheduling requirements or plain laziness may cause the reliability plans to be disregarded. This should, of course, be avoided. This is a management problem, and is beyond the scope of the tutorial.

3.4. Test Phase

Testing is intended to remove faults from the system before they become visible to the user. At this point, the data collection techniques planned out during the design phase will need to be in place, and will need to be used. Reliability growth models can be used to help decide how much testing is needed (when testing can be terminated) and to provide an indication of the expected reliability during operation.

3.5. Operation Phase

Once the system is in operation, the primary need from a reliability viewpoint is continued monitoring. This is where the failure log becomes important. Over time, as bugs are fixed and the system is changed to cope with changing business conditions, the software becomes increasingly difficult to maintain. Eventually, it reaches a stage where further repair or enhancement becomes ineffective; at this point, the software should be rewritten. By continuing to monitor failures, and to calculate the reliability of the system, it may be possible to discover when reliability begins to decrease in time for adequate planning for a replacement system.

4. FAULT TOLERANCE TECHNIQUES

4.1. General Aspects of Recovery

A fault tolerant computer system must be capable of recovering from failures (by definition). Since the failure is just a symptom of one or more underlying faults, which caused one or more errors, this recovery process must deal with these factors. Before giving design suggestions, let us examine this process.

4.1.1. Error Treatment

There are three aspects to coping with errors once a failure has occurred: error detection, damage assessment and recovery.

Error Detection. The success of all fault tolerance techniques is critically dependent upon the effectiveness of detecting errors. This activity is highly system dependent, so it is difficult to give general rules, except in the case of interface checks. The following suggestions are offered in [2].

- **Replication Checks.** It may be possible to duplicate a system action. The use of triply-redundant hardware [6] is an example. A calculation is carried out by at least three separate computers, and the results compared. If one results differs from the other two, that computer is assumed to have failed.

- **Timing Checks.** If timing constraints are involved in system actions, it is useful to check that they are satisfied. For process-to-process communications, for example, a time-out mechanism is frequently used.

- **Reversal Checks.** A module is a function from inputs to outputs. If this function is 1-1 (has an inverse), it may be possible to reproduce the inputs from the outputs, and compare the result to the original inputs.

- **Coding Checks.** Parity checks, checksums, cyclic redundancy codes, and the like can be used to detect data corruption. This is particularly important for communication lines and permanent file storage.

- **Reasonableness Checks.** In many cases, the results of a calculation must fall within a certain range in order to be valid. While such checks cannot guarantee correct results, they can certainly reduce the probability of undetected failures.

- **Structural Checks.** In a network of cooperating processes, it is frequently useful to perform periodic checks to see which processes are still running. Similarly, a periodic check on structural integrity is a good idea for complex data structures – a large distributed database, for example.

Damage Assessment. Once an error has been detected, it is necessary to discover the full extent of the damage. Calculations carried out using a latent error can spread the damage quite widely. Design techniques can be used to confine the effect of an error, and are therefore quite useful. One method is to carefully control the flow of information among the various components of the system. Note that these methods are almost impossible to retrofit to a system; this issue must be considered at the design stage.

Error Recovery. The final step is to eliminate all errors from the system state. This is done by some form of error recovery action. Two general approaches are available.

- **Forward Error Recovery.** In a few cases, a sufficient amount of correct state is available to permit the errors to be eliminated. This is, of course, quite system dependent. If a calculation was performed on incorrect input (spreading the damage), and the correct input can be recovered, the calculation can simply be redone. Forward error recovery can be quite cost effective when it is possible.

- **Backward Error Recovery.** If forward error recovery is not possible, one must restore the system to a prior state which is known to be correct. There are three general categories of such techniques, and they are usually used in combination.

  - **Checkpointing.** All (or a part) of the correct system state is saved, usually in a disk file.

  - **Audit Trails.** All changes that are made to the system state are kept in a transaction log. If the system fails, it can be reset to the latest checkpoint (or to the initial correct state), and the audit trail can be used to bring the system state

forward to a correct current state. This technique is frequently used in database management systems [11] and can also be quite effective in application systems. Careful planning is necessary.

  ▪ **Recovery Cache.** Instead of logging every change to a system, it is possible to incrementally copy only those portions of the system state which are changed. The system can be restored only to the latest incremental copy, unless an audit trail is also kept. Without the audit trail, all transactions since the latest incremental copy are lost. In some cases, this is quite sufficient.

• **Error Compensation.** This technique is possible if the erroneous state contains enough redundancy to enable the delivery of an error-free service from the erroneous (internal) state.

4.1.2. Fault Treatment

It may or may not be possible to continue operation after error recovery. Unless the fault was transient, however, it must be treated sooner or later. Nelson and Carroll [12] suggest four stages to fault treatment.

1. **Fault Location and Confinement.** One begins by isolating the fault by means of diagnostic checking (or some other technique). Whether or not the fault is immediately repaired after it has been located, it is generally wise to limit its effects as much as possible, so that the remainder of the system can be protected. Where high availability is required, the component containing the fault may be replaced so that operation may continue; in such cases, the next two steps take place off-line.

2. **Fault Diagnosis.** Once the fault has been isolated, it will be necessary to uncover the exact cause. This could be a hardware operational fault, a software bug, a human mistake, or something else. The technique to use, of course, depends on the source of the fault.

3. **System Repair and Reconfiguration.** If there is sufficient redundancy, this can take place during (possibly degraded) operation. Otherwise, the system may be stopped and repaired or reconfigured.

4. **Continued Service.** The repaired system is restored to a working condition, and restarted.

4.2. n–Version Programming Technique

The two principal software design techniques that have been developed for fault tolerant programming are the Recovery Block Technique and the N–Version Programming Technique [13]. Both use redundancy in order to recover from certain types of failures, but in different ways.

In the n–Version Programming Technique, a program specification is given to n different programmers (or groups of programmers), who write the programs independently. Generally, n must be at least 3. The basic premise is that the errors programmers make are independent: consequently, the multiple versions are unlikely to go wrong in the same way. In execution, all versions are executed, and voting is used to select the preferred

answer if there is disagreement. Whenever possible, different algorithms and programming languages are used. in the belief that this will further reduce the likelihood of common errors.

There are a number of problems with this approach. To begin with. the cost of programming is multiplied. since several versions are being written. Errors in the specification, of course, will not be detected. Additional design is required to prevent serious errors in any of the versions from crashing the operating system. and to synchronize the different tasks in order to compare results. Code must also be written to do the comparison. Finally, there is some evidence that programmers do indeed make the same sorts of mistakes, although this point is a bit controversial. Only the last point, if it proves to be true. is a fatal flaw, however.

4.3. Recovery Block Technique

The Recovery Block Technique uses redundancy in a different way. Here, there is just one program. but it incorporates algorithms to redo code that proves to be in error. The program consists of three parts. There is a primary procedure which executes the algorithm. When it has finished, an acceptance test is executed to judge the validity of the result. If the result is valid, that answer is passed back as the correct answer. However, if the acceptance test judges the answer to be incorrect, an alternative routine is invoked. This alternative will generally be simpler to code, but is likely to run slower, or use a different algorithm. The idea can be nested, of course.

The effect is to have a program module structured as follows:

answer = primary algorithm

```
if answer is valid, return (answer)
else {       answer = second algorithm
             if answer is valid, return (answer)
             else {      answer = third algorithm
                   .....

                    }
      }
```

Most of the time, only the first algorithm will be required. The result of failure is simply to slow down the execution, so this technique may not always be usable.

There is some evidence that this idea does indeed work. However. most of the criticisms offered for the n–Version Programming Technique would seem to apply here as well. in one form or another.

5. RELIABILITY MODELING AND MEASUREMENT

5.1. Computer System Reliability

From the viewpoint of the user, reliability is a system property. Measuring and modeling availability and reliability, therefore, needs to be done at the system level. A model at this level includes software, hardware, people and those portions of the environment that interact with the computer system. Power failures, fire and flood are examples of the latter.

The general approach is to recursively partition such a system into parts whose failures are independent. For example,

system = {*hardware, software, people, environment*}
hardware = {*cpu's, disks, communications*}
software = {*operating system, DBMS, terminal control, application*}

A failure rate is calculated for each. The serial system failure rate is just the sum of the individual failure rates:

$$\lambda_{system} = \sum_{i=1}^{n} \lambda_{part_i}.$$

This works provided (1) the system is in steady state, (2) failures of parts of the system are truly independent, and (3) the failure of any one part will result in system failure. Redundancy is the general approach to improving reliability, particularly in hardware. If a subsystem is partitioned into parts in such a way that all parts must fail before the subsystem fails, the failure rate calculation for parallel subsystems must be used.

$$\frac{1}{\lambda_{subsystem}} = \sum_{i=1}^{n} \frac{1}{\lambda_{part_i}}$$

These results would then be used in the earlier calculation when λ_{system} is being calculated.

If failures of the parts are not independent, or failure rates change over time, then Markov techniques frequently become necessary. This analysis is much harder.

General modeling of computer system failures was described briefly by the author at the 1988 CMG Annual Meeting [14]. For additional information, see the references given there or two recent books on computer system reliability, [15, 16].

The remainder of this section concentrates on application software reliability measurement and modeling. To begin with, all software faults are design faults. Depending on the circumstances, such faults may be repaired immediately or at some future time. Thus, software must be considered in two different ways.

The first way applies primarily to software produced by a vendor and sold to customers. In this case, the software can be considered an unchanging product (between releases, at least) by the customer. Thus, its failure rate is fixed during the lifetime of any particular release. This application product is usually independent (in the reliability sense) of the rest of the customer's system, and failure of the application software will result in system failure. Thus, the conditions for the serial model introduced at the beginning of Section 5.1 are satisfied, and the system failure rate can be easily calculated once the failure rates of the parts are known.

In-house application software generally has very different failure characteristics. Here, bugs are fixed as soon as they are discovered. The application may also be undergoing frequent enhancements and other modifications to fit the changing needs of the company. Payroll systems are an example. In this case, the application system is really a sequence of closely related but not identical systems, whose failure rates change over time. Obtaining reliable statistics on failures under these circumstances can be a real problem, since the system doesn't remaining stationary long enough to see more than a few failures. The solution is to use a Reliability Growth Model for Software. This is discussed in Section 5.2.

In either case, calculating reliability requires accurate data on failures. This is not easy, but is possible. This problem is discussed in Section 5.3.

5.2. Reliability Growth Models

Let us assume we have a software system S. Failures occur at (execution) times t_1, t_2, ..., t_n. After each failure, the fault that caused the failure may be fixed; thus, there is a sequence of programs $S = S_0, S_1, S_2, ..., S_n$, where S_j represents a modification of S_{j-1}, $1 \leq j \leq n$. (If the bug couldn't be found before another failure occurs, it could happen that $S_j = S_{j-1}$.)

A technique was developed several decades ago in the aerospace industry for modeling hardware reliability during design and test. Such a model is called a *Reliability Growth Model*. A component is tested for a period of time, during which failures occur. These failures lead to modifications to the design or manufacture of the component; the new version then goes back into test. This cycle is continued until design objectives are met. A modification of this technique seems to work quite well in modeling software (which, in a sense, never leaves the test phase).

Figure 1 shows some typical failure data, taken from [16, p. 305], of a program running in a user environment. In spite of the random fluctuations shown by the data (dots in the figure), it seems clear that the program is getting better – the time between failures appears to be increasing. This is confirmed by the solid curve, showing a five point moving average.
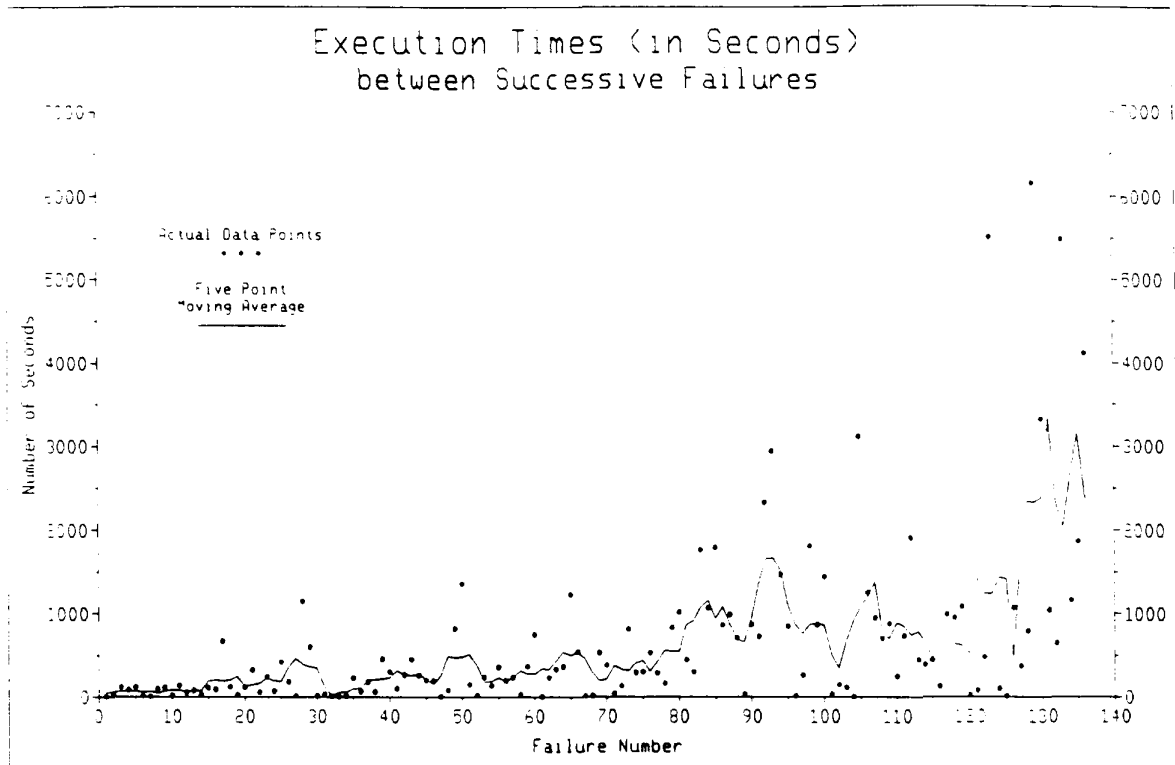
Figure 1. *Execution Time between Successive*
Failures of an Actual System

A reliability growth model can be used on data such as shown in the figure to predict future failure rates from past behavior of the program, even when the program is continually changing as bugs are fixed. There are at least three important applications of an estimate of future failure rates:

- As a general rule, the testing phase of a software project continues until personnel or money are exhausted. This is not exactly a scientific way to determine when to stop testing. As an alternative, testing can continue until the predicted future failure rate has decreased to a level specified before testing begins. Indeed, this was an original motivation for the development of reliability growth models.

  When used for this purpose, it is important to note that the testing environment is generally quite different from the production environment. Since testing is intended to force failures, the failure rate predicted during testing should be much higher than the actual failure rate that will be seen in production.

- Once into production, the failure rate can be monitored. Most software is maintained and "enhanced" during its lifetime: monitoring failure rates can be used to judge the quality of such efforts. The process of modifying software inevitably perturbs the program's structure. Eventually, this decreases quality to the point that failures occur faster than they can be fixed. Monitoring the failure rate over time can help determine this point, in time for management to make plans to replace the program.

• Some types of real world systems have (or ought to have) strict legal requirements on failure rates. Nuclear reactor control systems are an example. If a control system is part of the larger system, the failure rate for the entire system will require knowledge of the failure rate of the computer portion.

A variety of reliability growth models have been developed. These vary according to the basic assumptions of the specific model; for example, the functional form of the failure intensity. Choice of a specific model will depend, of course, on the particular objectives of the modeling effort. Once this is done, and failure data is collected over time, the model can be used to calculate a point or interval estimate of the failure rate. This is done periodically; say, after every tenth failure. A few of these models are discussed next. See [1, 16] for more information.

One of the most critical modeling assumptions is the kind of time that is used. Execution time and clock (calendar) time are the usual choices. Only the former is used here.

5.2.1. Duane Model (1964)

The earliest reliability growth model was proposed by Duane in 1964 in connection with hardware failures, and has sometimes been used successfully for predicting software reliability. This simple model assumes that the failure rate at time t can be given as

$$\lambda(t) = \alpha \beta t^{\beta-1}.$$

Knowing the times t_i that the first m failures occur permits maximum likelihood estimates of α and β to be calculated. If $\beta < 1$, reliability is improving. If $\beta > 1$, then reliability is getting worse; more errors are being added than removed. The number of bugs that can be expected to be found by time t is given by

$$m(t) = \alpha t^{\beta}.$$

In some cases, this model has proved to be quite successful; in other cases, the reverse has been reported.

5.2.2. Musa Model (1975)

This model begins by assuming that all software faults (bugs) are equally likely to occur, and are statistically independent. After each failure, the cause is determined and the software is repaired. (The bug is fixed.) Execution does not begin again until after this has happened; since execution time is the time parameter being used, this means (in effect) that bug fixing happens instantaneously. It is assumed that no new bugs are added during the repair process.

Consequently, the failure rate takes on an exponential form:

$$\lambda(t) = \alpha n e^{-\alpha t},$$

where the software originally had n bugs and α is a parameter relating to the failure rate of a single fault. Integrating gives the number of bugs found by time t:

$$m(t) = n(1 - e^{-\alpha t}).$$

The parameters can be estimated in standard ways after a number of bugs have been found and corrected.

This model has been reported to give good results. The primary difficulty is the assumption that all bugs are equally likely to occur. This seems unreasonable. In practice, some bugs seems to occur much more often than others; indeed, you would expect that the most common bugs are seen first. This line of reasoning gives rise to the next model.

5.2.3. *Littlewood Model* (1981)

Suppose that the program has n faults (bugs) when testing begins. Each of these faults will cause a failure after some period of time which is distributed exponentially and independently from any other fault. Instantaneous debugging is assumed. We assume that failures occur at times t_1, t_2, ..., t_i. After the ith bug has been found, Littlewood gives a failure rate of

$$\lambda(t) = \frac{(n-i)\,\alpha}{\beta + t_i + t}, \qquad\qquad t_i < t < t_{i+1},$$

where α and β are parameters to be determined by maximum likelihood estimates. The expected number of failures by time t is given by

$$m(t) = (n-i)\,\alpha \ln\left[\frac{\beta + t_i + t}{\beta + t_i}\right].$$

Notice that this function is has a step after each failure; the failure rate decreases abruptly after each bug is found. This is supposed to reflect the fact that more common bugs are found first. Like the Musa Model, this model is reported to give good results.

5.2.4. *Musa–Okumoto Model* (1984)

This model (also known as the logarithmic Poisson execution time model) is like the Littlewood model in that the failure rate function decreases exponentially with the number of failures experienced. The basic assumptions are the same as those for the Littlewood model. Here, we have

$$\lambda(t) = \frac{\alpha}{\alpha\,\beta\,t + 1}.$$

where α is the initial failure rate and β is a parameter. The expected number of failures by time t is given by

$$m(t) = \frac{1}{\beta}\ln(\alpha\,\beta\,t + 1).$$

Musa reports good results.

5.3. Data Collection Problems

Models, of course, are of no practical use if data is not available to use in estimating the parameters of the model. Reliability modeling is no different from performance modeling in this respect, except that data collection appears to be harder. There are problems with definitions, as well as problems with the actual data collection. There is a good discussion of this in reference [1], which should be read (and absorbed) before beginning any sort of data collection activity. Here, by way of example, I will discuss one definitional problem and one practical collection issue.

5.3.1. Defining Time

Reliability (failure rate) is measured in terms of "estimated failures per unit of time": consequently, the units in which time is measured have quite an effect on how failure rate is defined. The reference cites nine different ways of measuring time, which apply in different circumstances. The important issue is to choose one unit, and stick to it. Collecting data using one unit of time, and attempting to apply that to a model requiring a different unit is a bit suspect, to say the least. Here are the nine methods.

1. Clock (calendar, elapsed) time. This can be used when the application is run for the same amount of time each week, month, or whatever. For example, if your computer is operational only during prime shifts, five days a week, then elapsed number of weeks is a possible time unit.

2. Powered-up time. For applications that are always running when the computer is operational, but for which this time may vary from week to week, one can use the amount of time that the system was actually operational. Examples of such are operating systems, real-time control systems, and many transaction-based systems.

3. Normalized computer time. If your application is running on many different identical computers, you can pool the powered-on time from the various machines to form a combined run time. If the machines are of different powers, a normalization factor will have to be used to make the times commensurate.

4. Batch elapsed time. For batch programs in a multiprogramming system, you may wish to use the amount of clock time between program load and program termination.

5. CPU time. Programs used occasionally, or at varying frequencies, cannot easily be timed by an external clock. In this case, CPU time may be a good choice. In effect, you are counting the number of machine cycles devoted to that application.

6. Session time. Under some circumstances, you may be interested in the clock time for an interactive program, using the time from session initiation to session termination. Typically, this would be the case for a system in which the user's time is considered the important part of the application.

7. Message count. For distributed applications which exchange messages at more-or-less regular intervals, the number of message exchanges can be counted and used as an estimate of time.

8. Machine instruction count. How many object instructions were executed? This might be useful in cases where the same program is running on computers with the same instruction set, but much different cycle times. If you wish to compare experiences on such different machines, some method of eliminating the internal timing differences is necessary. Instruction count is one method of doing this.

9. Source instruction count. If there is no way to count actual instructions executed, counting the source instructions executed might serve as an estimator.

My personal choice is to use CPU time as the primary time unit unless there is overwhelming need for one of the others. Normalization is required if the CPUs are of different power.

5.3.2. A Common Data Collection Problem

Calculating the parameters of the different models will require an accurate record of the run time for the application (using whichever time unit you choose), plus a record of every single failure. This latter record must include the time of failure (in the chosen time units), plus a diagnosis of the actual fault that caused the failure. The latter needs to be quite specific, since the models generally require knowledge as to whether the failure is the first occurrence of the fault or a subsequent occurrence.

Humans cannot be relied on to accurately provide this information. If your application has been sold to customers, they have little incentive to provide the detailed information required. If used within your own shop, the operators are usually too busy to record the information, or forget to write down the time a fault occurred (due to the need to return the system to service), or lose patience in recording the fifty-first occurrence of a particular failure during a three hour period of time. If you are trying to collect data during application testing, development pressures are likely to cause failure recording to be dropped.

Thus, any recording needs to be automated in some fashion. The operation system accounting log may provide sufficient information to enable you to acquire run times. It might be possible to use the system trouble log to record information about application failures. Best of all, you may be able to have the application system itself record information about its running and any failures. (A monitor may be necessary for the latter, to allow for program crashes.) Some manual recording of failures will almost certainly still be necessary, but reducing this to an exceptional circumstance gives you a good chance to actually get the data.

6. CONCLUSION

The purpose of this tutorial has been to introduce the topic of software reliability engineering. As application development becomes increasingly an engineering activity, we can expect to see increasing emphasis on practical reliability engineering. Many of the problems involved are similar to the problems encountered in software performance engineering: the need for data collection, the need to become involved throughout the application life cycle, the need to educate programmers, users and management, and so on.

REFERENCES

1. *Software Reliability. State of the Art Report* **14:2**, Pergamon Infotech (1986).

2. Anderson, T. "Fault tolerant computing", in *Resilient Computing Systems*, T. Anderson (ed), Wiley (1985), 1–10.

3. Laprie, Jean–Claude. "Dependable computing and fault tolerance: Concepts and terminology", *The 15th Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society Press (1985), 2–11.

4. Siewiorek, Daniel P., and Robert S. Swarz. *The Theory and Practice of Reliable System Design*, Digital Press (1982).

5. Smith, David J. *Reliability Engineering*, Pitnam (1972).

6. Kopetz, H. "Resilient real-time systems", in *Resilient Computing Systems*, T. Anderson (ed), Wiley (1985), 91-101.

7. Anderson, Thomas, and John C. Knight. "A framework for software fault tolerance in real-time systems", *IEEE Trans. Soft. Eng.* **SE–9**, 3 (May 1983), 355-364.

8. Tsichritzis, Dionysios C., and Frederick H. Lochovsky, *Data Models*, Prentice-Hall (1982).

9. Page–Jones, Meilir. *The Practical Guide to Structured Systems Design*, Yourdon Press (1980).

10. L'Ecuyer, Pierre, and Jacques Malenfant. "Computing optimal checkpointing strategies for rollback and recovery schemes", *IEEE Trans. Computers* **37**, 4 (April 1988), 491-496.

11. Date, C. J. *An Introduction to Database Systems*, vol. 2, Addison-Wesley (1983).

12. Nelson, Victor A., and Bill D. Carroll. "Introduction to fault-tolerant computing", in *Tutorial: Fault-Tolerant Computing*, Nelson and Carroll (eds), IEEE Computer Society Press (1987), 1–4.

13. Pradhan, D. J. (ed), *Fault Tolerant Computing: Theory and Practice*, Prentice Hall (1986).

14. Lawrence, J. D. "Modeling the reliability of a real–time systems", *CMG '88*, (Dec. 12–16, 1988), 51–57..

15. Dhillon, B. S. *Reliability in Computer System Design*, Ablex Pub. Co. (1987).

16. Musa, J. D., A. Iannino and K. Okumoto, *Software Reliability: Measurement. Prediction, Application*, McGraw–Hill (1987).

DO NOT MICROFILM
COVER

A Tutorial on Software
Reliability Engineering

J. Dennis Lawrence

Received by OSTI

JUN 0 1 1989

Lawrence
Livermore
National
Laboratory