MODELING THE RELIABILITY OF A REAL-TIME SYSTEM

J. Dennis Lawrence
Lawrence Livermore National Laboratory

The world we live in is imperfect.
Ernst G. Frankel, 1984

ABSTRACT[1]

Under some circumstances, computer controlled systems can be quite dangerous. One of the tools used to analyze such situations is the construction of reliability and safety models. Four types of models are described briefly: reliability block diagrams, fault trees, Markov models and reliability growth models. There are many analogies to performance models, and the expertise of CMG members could be usefully applied to computer system reliability problems in addition to performance problems. This paper is an introduction to reliability modeling.

1. INTRODUCTION

Unreliable computer-controlled systems can kill. The following items were extracted from recent issues of *Software Engineering Notes*.

- "The Therac 25 linear accelerator is programmed to switch on command from high-intensity X-ray mode to low-intensity electron beam mode. To avoid the retyping of lengthy commands, editing of previous commands was permitted. Sloppy software permitted partial editing, with overdoses resulting from an incomplete switchover.... Two deaths and one serious injury resulted." [1]

- "The crash of an MD80, Northwest Flight 255, killing 156 in Detroit on 17 August 1987, had computer system and human implications. The investigation confirmed ... that the flaps were not set for takeoff.... The pilot and copilot ... neglected to set the thrust computer indicator – which prevented the throttle from staying in place. When the attempted takeoff began, a computerized warning system should have announced that the flaps had not been set; however, it had no power and remained silent." Thus, there were two human errors and one power failure. [2]

- Korean Air Lines flight 007 was shot down on September 1, 1983, after straying into Soviet airspace. No one knows precisely what it was doing there, but speculation indicates that the crew may have entered coordinate data into the inertial navigation system (INS) modules incorrectly. The system is triply redundant; the crew must enter trip coordinates into each of the three modules. This would seem to provide ample error checking; on disagreement between the three sets of entries, an alarm sounds. However, "though it is strictly against airline policy to do so, at the touch of a button the crew can 'autoload' coordinates from one INS to another." That is, the user interface was so tedious that air crews could simply bypass the error checking procedure [3].

One thing these examples have in common is that the failures were system failures. The components, by and large, carried out actions for which they were designed. Where components did fail, the failures would not have been serious under ordinary circumstances. However, under the actual circumstances, the failures cascaded into a system failure, and death was the result.

Measuring reliability is as important to controlling computer systems as measuring performance. Reliability must be engineered into a system as part of the design. Retrofitting reliability into software is harder than retrofitting performance. Modeling techniques exist that can assist in reliability engineering. Software can be built to compensate for some types of system failures. There are many analogies between performance engineering and reliability engineering.

There are two main sections to the paper. Section 2 provides background, and gives some classifications for different types of faults and failures. Section 3 describes some of the modeling techniques and tools that are available for measuring and predicting reliability.

I pointed out in [4] that the reliability of a real-time system cannot be fully separated from the performance of that system. This coupling is not considered here.

2. BACKGROUND

2.1. Some Definitions

2.1.1. Basic Concepts

The words *fault*, *error*, and *failure* have a plethora of definitions in the literature. For this paper, I will use the following:

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. [5, 6, 7]

An error is an incorrect state of hardware, software or data resulting from a component failure, a software bug, physical interference from the environment, an 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 [7, 5]. If never activated, the latent error never becomes effective.

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 [5].

2.1.2. Reliability Measures

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

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 [7, 8]. Reliability is often given in terms of the *failure rate*.

[1]This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

$$\lambda(t) = -\frac{d \ln R(t)}{dt},$$

or the *mean time to failure*,

$$mttf = \int_0^\infty R(t)\, dt.$$

If the failure rate is constant, $mttf = 1/\lambda$.

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 [7, 8]. 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*,

$$\mu(t) = -\frac{d \ln (1 - M(t))}{dt}.$$

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.

Unfortunately, the term *reliability* has both a specific and a broad meaning. The specific meaning is given earlier in this section. The broader use is as a general term for (specific) reliability, availability, maintainability and safety, as defined above. Which is meant should be clear from context.

2.2. Classification of Faults and Failures

Faults and failures can be classified in a variety of ways. This becomes important in the context of modeling, since different classes of faults and failures may require different modeling techniques. Here, we list several classifications; see [9] for a more complete discussion.

2.2.1. Fault Classification

Faults can be classified by their persistence and their source. A fault is a *design fault* if it can only be corrected by redesign; an *operational fault* if it results from the breakage of a properly designed portion of the system; and a *transient fault* if it is intermittent.

Faults can occur in many places: hardware, software, data, system structure (topology), user (operator, end user, equipment), environmental, or unknown.

2.2.2. Failure Classification

Failures can be classified by their scope and their effect on safety. A failure is called *internal* if it can be entirely contained within the device or process in which it occurred; *limited* if the effect remains within the immediate device or process, but some other device or process is necessary to fully resolve the failure; and *pervasive*, otherwise.

The effect a failure may have on safety can vary from none to total. At one extreme, some failures have no effect on safety whatever: a failure of a computer game in an arcade is an example. At the other extreme, the failure of a flight control computer can be deadly.

2.3. Phases of Reliability Engineering

Anderson [10] and Laprie [5] point out that the reliability of a computer system is determined by three 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. Finally, since neither design nor test is perfect, and since some new failures will occur during operation due to operational 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.

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.

Fault Removal techniques are necessary to eliminate any faults which have survived the fault avoidance phase. Testing is the usual technique.

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 to at least warn the user that errors exist in the system.

Once a failure has been detected, both the error that was the immediate cause of the failure and the underlying fault must be repaired. From the software engineering standpoint, the first of these is the most important. Of course, nothing can be done by the software to overcome some types of failures (such as fire, flood, or power failure); other techniques such as geographically distributed loosely coupled systems and uninterruptible power supplies may be required.

This entire topic is considered at greater length in the references given early in the section.

3. RELIABILITY MODELS

The primary modeling technique for performance problems is, of course, the queueing network model. Some people also use varieties of Petri nets. When considering reliability, availability and safety, however, a wider variety of models is needed. There are two major reasons for this: differences in kind and in degree among reliability, availability and safety; and differences in kind and degree among the varieties of faults. The operational reliability of a nonrepairable system is much easier to model than the operational availability of a repairable system, for example. Operational faults in general require different techniques than design faults. Reliability and availability, on the one hand, can be contrasted with safety, on the other.

There are, of course, many overlaps in modeling techniques as well as many differences. Consequently, the analyst needs to be familiar with several techniques. Four are described here: reliability block diagrams, fault trees, Markov models, and reliability growth models. The descriptions are necessarily simplified to fit within the confines of this paper.

Reliability modeling generally requires computer assistance, just as the simulation of queueing network models is best not attempted by hand. Several modeling packages are mentioned as illustrative of the various techniques. Mention of a product shall not be considered as any form of endorsement.

3.1. Reliability Block Diagrams

Reliability block diagrams are easy to construct and analyze, and are completely adequate for many cases involving the operational reliability of simple systems. See references [11 or 12] for a more complete discussion. We consider a system $\mathcal{S}$ composed of n components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$. Each component will continue to operate until it fails; repair is not carried out. The question is: what is the reliability of the entire system?

Let us suppose that component $\mathcal{C}_j$ has constant failure rate λ_j (and, therefore, a mttf of $1/\lambda_j$). The reliability of the component at time t is given by $R_j(t) = e^{-\lambda_j t}$. For example, a computer system might consist of a CPU with failure rate .23 (per thousand hours), a memory with failure rate .17, a communication line with failure rate .68 and an application program with failure rate .11. For these components, the mean time to failure is, respectively, 6 months, 8 months, 2 months and 12 months.

In the reliability block diagram, blocks represent components. These are connected together to represent failure dependencies. If the failure of any of a set of components will cause the system to fail, a series connection is appropriate. If the system will fail only if all components fail, a parallel connection is appropriate. More complex topologies are also possible.

In our example, this system will fail if any of the components fail. Hence, a series solution is appropriate, as shown in Figure 1.

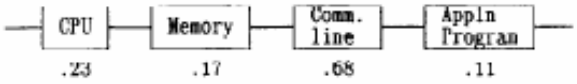

Figure 1. *Reliability Block Diagram of a Simple System*

The failure rate for a series system is equal to the sum of the failure rates of the components:

$$\lambda_S = \sum_{i=1}^{n} \lambda_i$$

For the example, the failure rate is 1.19 per thousand hours, giving a mttf of 840 hours (1.15 months). Notice that this is significantly smaller than the mttf for the least reliable component, the communication line.

The failure rate for a parallel system is more complex:

$$\frac{1}{\lambda_S} = \sum_{i=1}^{n} \frac{1}{\lambda_i}$$

Considering only the communication line, suppose we duplex or triplex this. Communication line failure rates are now .68, .34 and .23, respectively, yielding a mttf of 2 months, 4 months and 6 months. The reliability block diagram for just the communication line portion is shown in Figure 2.

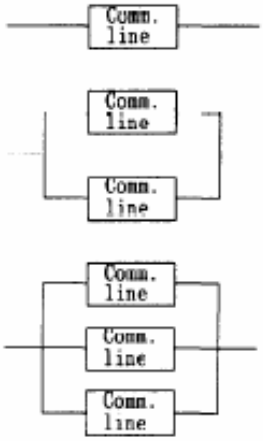

Figure 2. *Reliability Block Diagram of Single, Duplex, and Triplex Communication Line*

If we now insert the duplex communication line into the original system reliability block diagram, we have Figure 3.

This improves the failure rate to .85 per thousand hours, for a mttf of 1.6 months. Triplexing the communication line gives a failure rate of .74 per thousand hours, improving the mttf to 1.85 months.

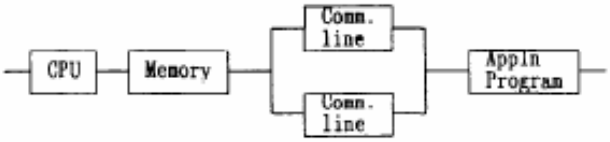

Figure 3. *Reliability Block Diagram of a Simple System with a Duplexed Communication Line*

The connecting lines in a reliability block diagram reflect failure dependencies, not any form of information transfer (although these are sometimes the same). The last diagram is interpreted to mean "failure of the system will occur if the CPU fails, if the memory fails, if both communication lines fail, or if the application program fails." Failures in real systems are sometimes more complex than can be represented by simply building up diagrams from series and parallel parts. Figure 4 gives an example.

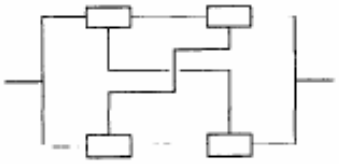

Figure 4. *Reliability Block Diagram that Cannot Be Constructed from Serial and Parallel Parts*

Analysis of a reliability block diagram is, in the most general cases, rather complex. However, if the graph can be constructed from series and parallel components, the solution is quite easy. Salmer [13] describes a program, SPADE, that can solve such problems efficiently.

The advantages of the reliability block diagram are its simplicity and the fact that failure rates can frequently be calculated rather simply. A simple technique, of course, is applicable only to simple forms of reality. Severe limitations are the assumptions that failures of components are statistically independent and that failure rates are constant over time. Complex nonrepairable systems are better analyzed using a fault tree, while analysis of repairable systems requires the use of a Markov model.

3.2. Fault Tree Models

The use of fault tree models has developed out of the missiles, space and nuclear power industries. Recent introductions can be found in [12 and 14].

One begins by selecting an undesirable event, such as the failure of the computer system. In more complex systems that use computers as subsystems, the failure could involve portions of the larger system as well: meltdown of the reactor, incorrect targeting of a missile, failure of the environmental control of a space capsule and the like.

The fault tree is developed by successively breaking down events into lower-level events that generate the upper level event; the tree is an extended form of and-or tree. See Figure 5.

This diagram means that event E1 occurs only if both of events E2 and E3 occur. E2 can occur if either of E4 or E5 occur. Event E3 will occur if any two of E6, E7 and E8 occur. Fault trees are generally constructed using AND, OR and R-of-N combinations.

The fault tree is expanded "downwards" until events are reached whose probability can be given directly. Note the assumption that the occurrence of the events at the bottom of

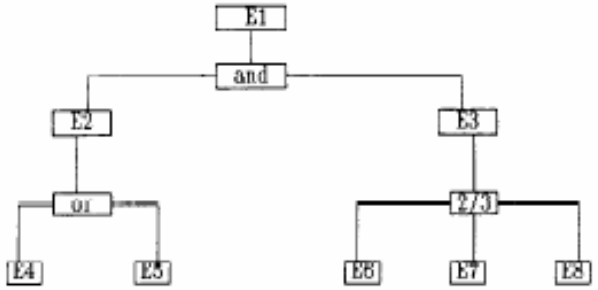

Figure 5. *Simple Fault Tree*

the tree are mutually independent. In many cases, the actual probabilities of these events are estimated (or simply guessed); this is particularly true if they represent human failures, uncontrollable external events and the like. Note that, in general, the same event may occur several times at the lowest level, if it can contribute to the main failure in several ways.

The fault tree can be evaluated from bottom to top. Consider the tree shown in Figure 6, in which the lowest level events are not replicated. Suppose $p_j(t)$ denotes the probability that event E_j will occur by time t, and we wish to evaluate an AND node.

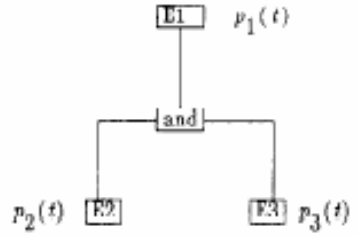

Figure 6. *And Node Evaluation in a Fault Tree*

Here, $p_1(t) = p_2(t)\ p_3(t)$; probabilities at AND nodes multiply. If we have an OR node, as shown in Figure 7, we have $p_1(t) = 1 - (1 - p_2(t))(1 - p_3(t))$. Generalization to AND and OR nodes with more than two events should be clear. An R-of-N node represents a Boolean combination of AND and OR nodes, so its evaluation is straightforward, though tedious.

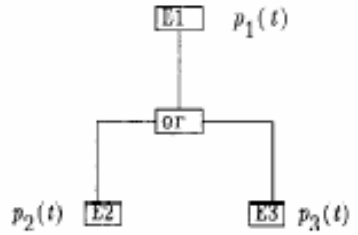

Figure 7. *Or Node Evaluation in a Fault Tree*

Reliability block diagrams and fault trees represent the same logic, so they can be converted from one form to the other. Which to choose for an actual problem is a matter of convenience.

Then, why was the block diagram analyzed in terms of failure rates and mttf, while the fault tree was analyzed in terms of probability of failure? These are, of course, connected, so the

$$\text{mttf} = \frac{1}{\lambda} \qquad \text{(where } \lambda \text{ is the failure rate of the component)}$$

$$R(t) = e^{-\lambda t}$$

$$R(t) = 1 - p(t)$$

In practice, fault trees tend to have thousands of basic events, and replication of basic events is common. Analysis of such a tree requires computer assistance. You will not be surprised to learn that such programs are available. For example, Fco [15] describes a program called PAFT F77 that calculates the failure rate for the top (and all intermediate) nodes of a fault tree much as described in this section.

Some minor generalizations of reliability block diagrams and fault trees are possible; in particular, common cause failures (events that cause several components to fail) can be handled with some difficulty. Such are beyond the scope of this introductory paper.

3.3. Markov Models

Markov models are used in performance analysis to capture the idea of system state, and a probabilistic transition between states. These models are used in reliability analysis for the same reasons. Here, the *state* represents knowledge of which components are operational and which are being repaired (if any). See [12] for a more complete introduction.

Straightforward block diagram models and fault tree models can be evaluated using Boolean algebra, as described in the last two sections. More complex models are generally translated into Markov models first. Systems whose components are repairable, and systems where component failures have interactions, are usually modeled directly by Markov models, with cycles. A number of examples are given here.

Throughout this section, we consider a system $\mathcal{S}$ with components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$; there may be more than one instance of each component. Component $\mathcal{C}_j$ has constant failure rate λ_j and constant repair rate μ_j.

Let us begin with a system which contains three CPUs. Only one is required for operation; the other two provide redundancy. Only one repair station is available, so even if more than one CPU is down, repairs happen one at a time. If state k is used to mean "k CPUs are operating," the Markov model is shown in Figure 8.

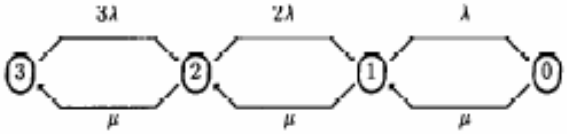

Figure 8. *A Simple Markov Model of a System with Three CPU's*

This has a striking resemblance to a performance model of a single queue with three servers and limited queue size and can be solved in the same way to yield the probabilities of being in each of the four states. The interpretation is that the system is operational except in state 0.

Now, suppose we add two memories to the system. Failure rates are λ_c and λ_m for the CPUs and memories, respectively, and similar notation is used for repair rates. We use a label of "k,l" for states, meaning "k CPUs and l memories are operational." The system is operational except in states "0,0", "0,1" and "1,0." The diagram is given in Figure 9.

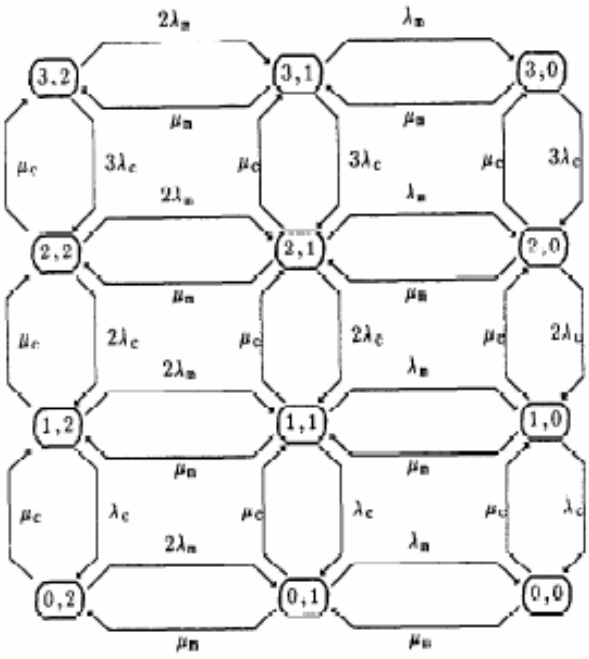

Figure 9. *Markov Model of a System with CPU's and Memories*

It is clearly possible to model the case where failure rates vary with circumstances. In our first example (with 3 CPUs), it might happen that each CPU has a failure rate of λ when all three are available, λ' if two are available and λ'' if only one is available. This might reflect a situation where the load is shared among all operational CPUs; increased load causes an increased failure rate for some reason: $\lambda < \lambda' < \lambda''$. The diagram is modified as shown in Figure 10.

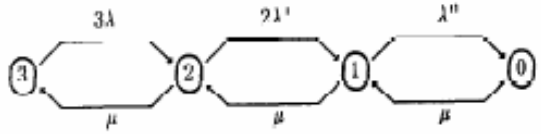

Figure 10. *Simple Markov Model with Varying Failure Rates*

It was pointed out in [1] that failure rates in computer systems vary over time. In particular, failures tend to be more frequent immediately after preventive maintenance; after this transition period, the failure rate will return to the normal value. Transient failures frequently show a similar pattern: a memory unit will show no failures for months; then, a day with hundreds of transient faults will occur; the next day, the situation is back to normal, even though no repair was done. Such situations cannot be modeled with reliability block diagrams or fault trees, but can easily be modeled using Markov chains. The next example shows four states, modeling the memory problem. State 1 is the normal operational state and state 0 represents a "hard" memory failure. The failure rate from state 1 is λ and the memory repair rate from state 0 is μ.

There is, however, a (very small) chance of changing to state 3, where frequent transient memory errors are possible. Once there, memory faults occur with rate $\lambda' >> \lambda$. Since these are transient, "repair" happens very rapidly (within milliseconds, or faster). Eventually, the system returns to state 1 and normality resumes. Note that hard failures can also occur in state 3; I have assumed that the process of repairing these will abort the period of transient errors, so I have made a transition to state 0. Other models are possible, of course. See Figure 11.

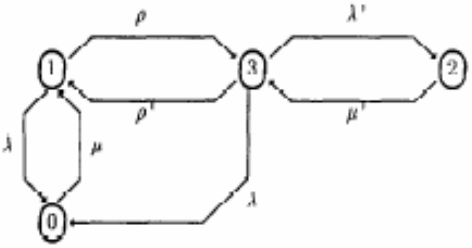

Figure 11. *Markov Model of a Simple System with Transient Failures*

It is easy to see that a computer system with thousands of components (hardware, software, users and so on) can lead to very complex models. Indeed, Markov models with tens to hundreds of thousands of states are not unknown. Obviously, solving such a large problem requires computer assistance. Many such programs have been developed; the following list of recently developed reliability evaluation systems is incomplete.

- ARIES 81 (the Automated Reliability Interactive Estimation System) was developed at UCLA in 1981. See [16]; a list of reliability evaluation tools developed during the 1970's is contained in this reference.
- CAST (the Complementary Analytic-Simulative Technique) was developed for use in judging the fault tolerance of the shuttle-orbiter data processing system [17]. The underlying modeling capability is apparently a special case of ARIES, but with some useful additional capabilities.
- SAVE (the System Availability Estimator) was developed at the IBM Watson Research Center [18] to solve availability and reliability models.
- CARE III (Computer-Aided Reliability Estimation program) was developed by Raytheon and others under contract to NASA in 1985 [19]. It represents an advance over the earlier systems, and remains in active development by NASA.
- HARP (the Hybrid Automated Reliability Predictor), developed at Duke University for NASA [20], represents a major advance. It decomposes a model into a fault-occurrence/ repair model and a fault/ error-handling model. The latter includes solution methods by versions of CARE III, ARIES, stochastic Petri nets, and others.
- SHARPE (the Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a recent development that divides a model into a hierarchy of models, and permits different kinds of models at different levels [21]. Seven model types are allowed: varieties of reliability block diagrams, fault trees, Markov chains, and directed graphs.

3.4. Reliability Growth Models

The previous types of models all implicitly assume that the system being modeled doesn't change. If, for example, we change the failure rate of a component (by replacing it with a new model), the reliability model must be re-evaluated.

That's OK for systems whose components remain unchanged for long periods of time, so sufficient faults occur to permit a failure rate to be determined. But, what about software?

To begin with, software and hardware faults are different in kind. As a general rule, all hardware faults that we users see are operational or transient in nature. All application software faults, on the other hand, are design faults. When a system failure is traced to a software fault (bug), the software is repaired (the bug is fixed). In a sense, we now have a new pro-

gram – certainly, the failure rate has changed. As a consequence, there are generally too few faults in a program to permit a failure rate to be calculated before the program is changed.

To be more precise: let us assume we have a software system $\mathcal{P}$. Failures occur at times $t_1, t_2, \ldots, t_n$. After each failure, the fault that caused the failure may be fixed; thus, there is a sequence of programs $\mathcal{P} = \mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, where $\mathcal{P}_i$ represents a modification of $\mathcal{P}_{j-1}$, $1 \leq j \leq n$. (If the bug couldn't be found before another failure occurs, it could happen that $\mathcal{P}_j = \mathcal{P}_{j-1}$.)

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 12 shows some typical failure data, taken from [22, 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.

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.

The original reliability growth model proposed by Duane in 1964 suggests that the failure rate at time t can be given as $\lambda(t) = \alpha \beta t^{\beta-1}$ [23]. Knowing the times t_i that the first m failures occur permits maximum likelihood estimates of α and β to be calculated:

$$\beta = m \div \sum_{i=1}^{m-1} \ln (t_m / t_i)$$

$$\alpha = m / t_m^{\beta}$$

Healy pointed out that the Duane model is sensitive to early failures, and suggested a recursive procedure:

$$\tau_4 = .25 (t_1 + t_2 + t_3 + t_4)$$

$$\tau_m = (1 - w)\, \tau_{m-1} + w (t_m - t_{m-1}), \text{ for } m > 4,$$

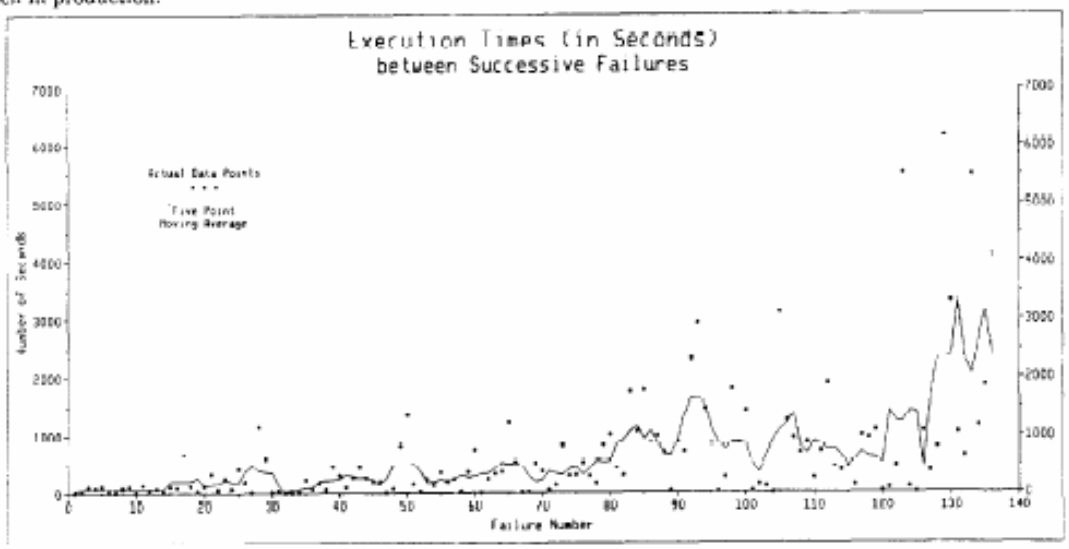

Figure 12. *Execution Time between Successive Failures of an Actual System*

yielding an estimate of the failure rate as

$$\lambda(t_m) = 1 / \gamma_m.$$

Healy recommends a weight of $w = .25$.

Using these two methods of calculating a failure rate (there are others) on the data of Figure 12 gives the following estimates of failure rate:

failure number	time (sec)	Duane	Healy
10	571	.014421	.015750
20	1986	.006730	.006881
30	5049	.003369	.002903
40	6380	.004079	.004414
50	10089	.003004	.001733
60	12559	.003028	.002602
70	16185	.002726	.002727
80	20567	.002409	.001711
90	29361	.001734	.001325
100	42015	.001236	.000885
110	49416	.001168	.001286
120	56485	.001133	.001615
130	74361	.000876	.000407

Software reliability models are not as mature as the other models described in this paper. They do appear very promising, and the need is certainly great. It appears that the models that have been developed so far are well worth using as part of the software development process.

Different kinds of models can be used to solve parts of a large problem. For example, a complex problem involving repairable and nonrepairable components might be modeled using a Markov technique. The nonrepairable portion, which could look like Figure 3, might consist of a submodel analyzed by a reliability block diagram. The application program portion of that could, in turn, be modeled by a reliability growth model. This use of combinations of modeling techniques is a powerful method of analysis.

4. CONCLUSION

CMG is "oriented toward practical...uses of computer performance....". So, why this paper on reliability? The two sets of problems are becoming increasingly intertwined, especially in real time systems and transaction processing systems. CMG members have great experience in practical problems involving performance – collection, evaluation and analysis of measurement data; construction and evaluation of models; real-world problems of cost, manpower, schedule and similar considerations; and the necessity of educating managers, users and fellow computer professionals. All of these issues arise as well in problems involving reliability and safety. It is my expectation that many of you will find yourselves involved in reliability problems in coming years, and that your practical knowledge can greatly assist in solving them.

REFERENCES

1. Neumann, Peter G. "Update on Therac 25", in "Risks to the public in computers and related systems", *Soft. Eng. Notes* 12, 3 (July 1987), 7.

2. Neumann, Peter G. "Computer failed to warn jet crew", in "Risks to the public in computers and related systems", *Soft. Eng. Notes* 12, 4 (October 1987), 2.

3. Jong, Steve. "More on Korean Air Lines Flight 007", in "Risks to the public in computers and related systems", *Soft. Eng. Notes* 12, 1 (January 1987), 5.

4. Lawrence, J. D. "Conceptual issues in measuring the behavior of a distributed real-process-control computer system", *CMG '87 Conference Proceedings*, Computer Measurement Group (1987), 474-481.

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

6. Randell, B. P. A. L., and P. C. Treleaven. "Reliability issues in computing system design", *ACM Computing Surveys* 10, 2 (June 1978), 123-165.

7. Siewiorek, D. P., and R. S. Swarz. *The Theory and Practice of Reliable System Design*, Digital Press (1982).

8. Smith, D. J. *Reliability Engineering*, Pitnam (1972).

9. Lawrence, J. D. "Software reliability engineering", *Structured Development Forum X* (August 1988), 31-11.

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

11. Lloyd, D. K., and M. Lipow. *Reliability. Management, Methods and Mathematics*, Prentice Hall (1977).

12. Pages, A., and M. Gondran. *System Reliability Evaluation and Prediction in Engineering*, Springer-Verlag (1986).

13. Sahner, R. A., and K. S. Trivedi. "Performance and reliability analysis using directed acyclic graphs", *IEEE Trans. Software Engineering* SE-13, 10 (October 1987), 1105-1114.

14. Henley, E. J., and H. Kumamoto. *Designing for Reliability and Safety Control*, Prentice Hall (1985).

15. Fen, T. "PAFT 77, program for the analysis of fault trees", *IEEE Trans. Reliability* R-35, 1 (April 1986), 48-50.

16. Makam, S. V., and A. Avizienis. "ARIES 81: A reliability and life-cycle evaluation tool for fault tolerant systems", *Proc. 12th IEEE Int. Symp. on Fault-Tolerant Computing* (1982), 267-274.

17. Geist, R. M., and K. S. Trivedi. "Ultrahigh reliability prediction for fault tolerant computer systems", *IEEE Trans. Computers* C-32, 12 (Dec. 1983), 1118-1127.

18. Goyal, A., W. C. Carter, E. de Souza e Silva, and S. S. Lavenberg. "The system availability estimator", *IEEE Annual Int. Symp. on Fault-Tolerant Comp. Systems* (July 1986), 84-89.

19. Stiffler, J. J. "Computer-aided reliability estimation", *Fault-Tolerant Computing, Theory and Techniques*, vol. 2, D. K. Pradhan, ed, Prentice Hall (1985), 633-657.

20. Bavuso, S. J., J. B. Dugan, K. S. Trivedi, E. M. Rothmann and W. E. Smith. "Analysis of typical fault-tolerant architectures using HARP", *IEEE Trans. on Reliability* R-36, 2 (June 1987), 176-185.

21. Sahner, R. A., and K. S. Trivedi. "Reliability modeling using SHARPE", *IEEE Trans. on Reliability* R-36, 2 (June 1987), 186-193.

22. Musa, John D., Anthony Iannino and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill (1987).

23. Healy, John. "A simple procedure for reliability growth modeling", *Proc 1987 Annual Rel. and Maint. Symp.* (1987), 171-175.