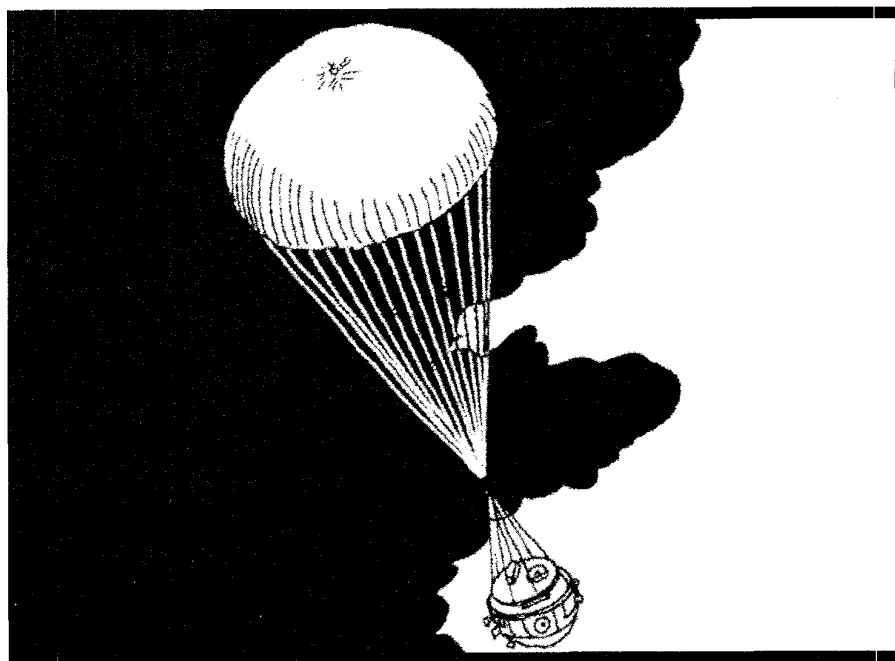


# A Generalized Technique for Simulating Software Reliability

Although several models have been proposed for assessing software reliability, none has emerged as the most effective predictor. The authors offer a general simulation technique that relaxes or removes many of the usual reliability-modeling assumptions and expands the reliability process to encompass the entire software life cycle.



ROBERT C. TAUSWORTHE  
Jet Propulsion Laboratory  
MICHAEL R. LYU  
AT&T Research

**S**oftware reliability has been the subject of wide study over the past 20 years. At least 40 different models have been published so far.<sup>1</sup> These studies have focused primarily on proposing, analyzing, and evaluating the performance of models that assess current reliability and forecast future operability from observable failure data, using statistical inference techniques. However, none of these models extends over the entire reliability process; most tend to focus only on failure observance during testing or operations. Moreover, none of these reliability models has emerged as the “best” predictor in all cases.<sup>2</sup>

Any of several factors may be responsible for this: oversimplification of the failure process, the quality of observed data, the lack of sufficient data to make sound inferences, and serious differences between the proposed model and the true underlying reliability process or processes. The basic nature of the failure processes may conceivably differ among individual software developments.

We propose a general simulation technique that relaxes or removes many of the usual reliability-modeling assumptions and expands the reliability process to encompass the entire software life cycle. Some of these assumptions are

- ◆ Testing or operations randomly encounter failures.
- ◆ Failures in nonoverlapping time intervals are independent.
- ◆ The test space “covers” the use space, or operational profile.

## RELIABILITY-PROCESS SIMULATION THEORY

The fundamental assumption of reliability-process simulation is that every stochastic event results from an underlying, instantaneous *conditional event-rate* random process.<sup>1</sup>

**Discrete-event simulation framework.** A conditional event-rate process is one for which the probability that an event occurs in the interval  $(t, t + dt)$ , given that it has not occurred prior to time  $t$ , is equal to  $\beta(t) dt$  for some function  $\beta(t)$ . The statistical behavior of this process is well-known<sup>2</sup>: The probability that an event  $\varepsilon$  will have occurred prior to a given time  $t$  is related by the expression

$$\text{Prob}\{\varepsilon \text{ occurs in } (0, t)\} = P(t) = 1 - \exp\left(-\int_0^t \beta(\tau) d\tau\right) = 1 - e^{-\lambda(0,t)} \quad (1)$$

When the events of interest are failures,  $\beta(t)$  is often referred to as the *process-hazard rate* and  $\lambda(0, t)$  is the *total hazard*. If  $\lambda(0, t)$  is known in closed form, the event probability can be analyzed as a function of time. But if many related events are intricately combined in  $\beta(t)$ , the likelihood of a closed-form solution for event statistics dims considerably. The expressions to be solved can easily become so convoluted that calculation of results requires a computer programmed with comparatively complex algorithms.

Of special interest here are discrete event-count processes that merely record the occurrences of rate-controlled events over time. The function  $\beta_n(t)$  denotes the conditional occurrence rate, given that the  $n$ th event has already occurred by the time  $t$ . The integral of  $\beta_n(t)$  is  $\lambda(0, t)$ . These processes are termed nonhomogeneous when  $\beta_n(t)$  depends explicitly on  $t$ . The probability  $P_n(t)$  that  $n$  events occur in  $(0, t)$  is much more difficult to express than Equation 1, and does not concern us here.

One important event-rate process is the discrete Markoff process. A Markoff process is said to be homogeneous when its rate function is sensitive only to time differences, rather than to absolute time values. The notation  $\beta_n(t)$ , in these cases, signifies that  $t$  is measured from the occurrence time  $t_n$  of the  $n$ th event.

When the hazard rate  $\beta_n(t)$  of a Markoff event-count process is independent of  $n$ , you may readily verify that the general event-count behavior is a nonhomogeneous Poisson process whose mean and variance are given by

$$\begin{aligned} \bar{n} &= \lambda(0,t) \\ \sigma^2 &= \lambda(0,t) \\ \frac{\sigma}{\bar{n}} &= 1/\sqrt{\lambda(0,t)} = 1/\sqrt{\bar{n}} \end{aligned} \quad (2)$$

The homogeneous, constant-event-rate Poisson process is described by  $\lambda = \beta t$ . Homogeneous Poisson-process statistics thus only apply to the homogeneous Markoff event-count process when the Markoff  $\beta_n(t) = \beta$  is constant.

Equation 2 shows that, as  $\bar{n}$  increases, the percentage deviation of the process decreases. In fact, any event process

with independence among events in nonoverlapping time intervals will exhibit relative fluctuations that behave as  $O(1/\sqrt{\bar{n}})$ , a quantity that gets increasingly smaller for larger  $\bar{n}$ . This trend signifies that Poisson and Markoff processes involving large numbers of event occurrences will tend to become, percentage-wise, relatively regular. If physical processes appear to be very irregular, then it will be impossible to simulate them using independent-increment assumptions with regular-rate functions.

In one sense, the NHPP form is inappropriate for describing the overall software-reliability profile. Software reliability grows only as faults are discovered and repaired, and these events occur only at a finite number of times during the life cycle. The true hazard rate presumably changes discontinuously at these times, whereas the NHPP rate changes continuously. In any case, the event-count Markoff model of software reliability is more general than the NHPP form, in that there is no assumption that its cumulative rate  $\lambda$  is independent of  $n$  or  $t_n$ .

**Multiple event processes.** Conditional event-rate processes are also characterized by the property that the occurrences of several independent classes of events,  $\varepsilon_1, \dots, \varepsilon_f$ , with rate functions  $\beta_n^{(1)}(t), \dots, \beta_n^{(f)}(t)$ , respectively, together behave as if  $f$  algorithms of the single-event variety were running simultaneously, each with its own separate rate function,  $\text{beta}[i](n, \tau)$ , controlling the  $n$ th occurrence of event  $\varepsilon_i$  at time  $t$ . That is, the event occurrence process is equivalent to a single event-rate process governed by its composite-rate function,

$$\beta_n(0, t) = \sum_{i=1}^f \beta_n^{(i)}(0, t)$$

When event occurrences in non-overlapping intervals are independent, each  $(t_n, t_{n+1})$  interval is governed by a nonhomogeneous Markoff process with rate  $\beta_n(t, t_n)$ .

$$\beta_n(t, t_n) = \sum_{i=1}^f \beta_n^{(i)}(t, t_n)$$

When a new event  $\varepsilon_i$  is added to or deleted from the distinguished class of events,  $\beta_n(t, t_n)$  readjusts to include or exclude the corresponding  $\beta_n^{(i)}(t, t_n)$  function and the simulation proceeds. This characteristic provides a simple and straightforward method to simulate the effects of fault and defect injections and removals.

## REFERENCES

1. N. Roberts et al., *Introduction to Computer Simulation*, Addison-Wesley, Reading, Mass., 1983.
2. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 1965.

- ◆ All failures are observed when they occur.
- ◆ Faults are immediately removed upon failure or are not counted again.
- ◆ Execution time is the relevant independent variable.

Many of these assumptions evoke controversy and require further qualification, so it is tempting to dismiss them. In particular, the second assumption can be made less restrictive: Faults produce independent failures.

The final four assumptions are not at

**Simulation can easily distinguish those faults that have been removed and those that have not.**

all necessary to the technique we present. The degree of commonality among test space and use space is rarely known, but can be modeled if needed. Simulation can mimic the failure to observe an error when it has occurred and can also mimic any system outage caused by an observed failure.<sup>3</sup> Furthermore, simulation can easily distinguish those faults that have been removed and those that have not, so multiple failures from the same unrecovered fault can be readily reproduced.

Finally, while execution time is pertinent to some life-cycle activities, it is not appropriate to others, such as inspections. Simulation can translate all model-pertinent times to wall-clock or calendar time by appropriate use of workload, computer, and resource schedules. This composite process is embodied in a Monte Carlo simulation tool, SoftRel,<sup>4</sup> which is available through NASA's Computer Software Management Information Center and

on the diskette that accompanies *Handbook of Software Reliability Engineering*.<sup>5</sup>

But of what interest is a reliability-process simulation tool to the software practitioner? One powerful way of understanding a pattern in nature is to recreate it in a simulation or other representative model. Because reliability is one of the first-cited indicators of quality, a tool that can reproduce the characteristics of the process that builds reliability offers a potential for optimization via trade-offs that involve scheduling, resource allocation, and the use of alternative technologies and methodologies. The parameters that characterize that process become metrics to be managed as means to achieve prescribed levels of quality.

A simulation tool may vary from simple to complex, depending on the scope of the process being modeled and the fidelity required by the user. Most analytic models require only a few inputs; SoftRel can use up to 70. Earlier models could report only a few facts about the unfolding process, but SoftRel can report up to 90. Of course, SoftRel can simulate the simple models as well.

The 70 input parameters are spread over the 14 activities that comprise the reliability process. Thus, each subprocess uses, on average, only five parameters, some of which quantify interrelationships among activities. Each of the activity submodels is thus fairly straightforward. You need not simulate the entire process all at once. If you only have data available on the failure and repair portions of the process, then you need input to the simulator only the parameters that characterize these activities.

At first, it may seem a daunting task to have to give values to all the development-environment parameters and produce a project-resource allocation schedule. But these tasks should

become progressively easier as experience locates the stable and volatile elements of the projects you undertake. Besides, we believe that the elements that characterize and control the process must be estimated anyway — whether the simulator uses them or not — to understand and manage the reliability process effectively. Parameters such as the expected fault density and the average defect-repair cost are familiar values extracted from prior project histories.

## SIMULATION BUILDING BLOCKS

The software-reliability process is a *conditional event-rate process*, which means that the probability that an event occurs in the interval  $(t, t + dt)$ , given that it has not occurred prior to time  $t$ , is equal to  $\beta(t) dt$  for some function  $\beta(t)$ . This process can be programmed in C like this<sup>6</sup>:

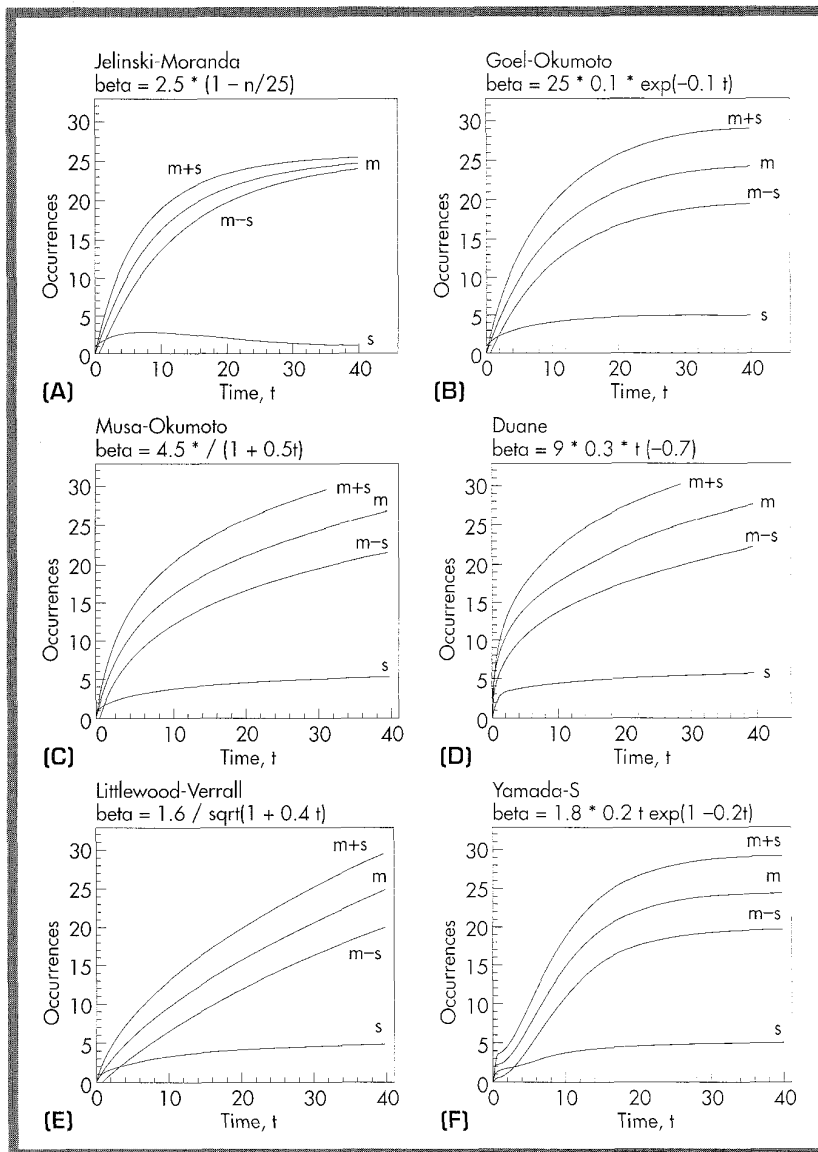
```

/* t and dt are assumed set
   prior to this point */
events = 0;
T = 0.;
while (T < t)
{
    T += dt;
    if (chance
        (beta(events, T)
         * dt))
        events++;
}
/* the event has occurred a
   number of times at
   this point */

```

The  $dt$  in such simulations must always be chosen such that the variations in the failure rate  $\beta(t)$  over the incremental time intervals  $(t, t + dt)$  are negligible, and such that  $\beta(t) dt < 1$ , so that the instantaneous event probability does not reach unity.

In the code segment above, `chance(x)` compares a  $(0, 1)$ -uniform `random()` value with  $x$ , thus attaining the specified instantaneous probability function. The form of `beta(events, T)` acknowledges that the event rate



**Figure 1.** Simulation results based on six software-reliability models. Each diagram shows the mean of the simulation results as the line marked "m"; the confidence intervals above the standard deviation as the line marked "m+s"; and the confidence intervals below the standard deviation as the line marked "m-s." The standard deviation along the time line is presented as the line marked "s" at the bottom.

function may change over time and may be sensitive to the number of event occurrences up to the current time. The computational complexity of this algorithm is  $O(\beta T/\Delta t)$ , in constant space. The  $\beta$  component represents the maximum time required to compute  $\beta_n(t)$ . Even today's moderately fast computers can easily handle this level of complexity.

The preceding simulation illustration is simple and yet very powerful. For example, some published analytic

models treat or approximate the overall reliability growth as a nonhomogeneous Poisson process in execution time, while others focus on Markoff execution-time interval statistics. Many of these differ only in the forms of their rate functions<sup>1</sup>:

◆ The Jelinski-Moranda model<sup>7</sup> deals with adjacent time-interval subprocesses in which  $\beta_n(t) = f(n_0 - n)$ , where  $n_0$  is the (unknown) number of initial faults and  $\phi$  is the per-fault failure rate.

◆ The Goel-Okumoto model<sup>8</sup> deals with overall reliability growth, in which  $\beta(t) = n_0 \phi e^{-\phi t}$ , where  $n_0$  and  $\phi$  are constant parameters. It can be shown that this model produces results very much like the Jelinski-Moranda model with  $n = n_0 (1 - e^{-\phi t})$ .

◆ The Musa-Okumoto model<sup>9</sup> also describes overall reliability growth, in which  $\beta(t) = \beta_0 / (1 + \theta t)$ , where  $\beta_0$  is the initial failure rate and  $\theta$  is a rate-decay factor. Both  $\beta_0$  and  $\theta$  are constant parameters.

◆ The Duane model<sup>10</sup> is another overall reliability-growth model, where  $\beta(t) = kbt^{b-1}$  and  $k$  and  $b$  are constant parameters.

◆ The Littlewood-Verrall inverse linear model<sup>11</sup> is an overall reliability-growth model with  $\beta(t) = \phi / \sqrt{1 + kt}$ , where  $\phi$  and  $k$  are constant parameters.

◆ The Yamada delayed S-shape model<sup>12</sup> is yet another overall reliability-growth model, with  $\beta(t) = \phi \gamma t e^{(1-\gamma)t}$ , where  $\phi$ , the maximum failure rate, and  $\gamma$  are constant parameters.

SoftRel can simulate any of these six models and use them as a basis for predicting the reliability of a given software project. Although these analytic models can be run using other commercially available software, because of their rigid, mathematical nature and their dependence on hard data, these models are of limited usefulness until late in a project's life cycle. SoftRel, on the other hand, uses functions and parameters to much more accurately simulate the reliability of a given software project. This makes SoftRel more useful earlier in the project life cycle.

Figure 1 shows the results obtained from simulating these six models and their underlying reliability process. Each of the simulation diagrams lists the rate function ( $\beta$ ) and its associated parameters. The parameters are set up such that there are initially about 25 faults in the system. We chose the value of 25 to emphasize the variability

of the failure processes at low fault rates; at higher fault concentrations, the decreasing  $\sigma/\bar{n}$  produces smaller deviations.

To simulate the occurrence of failure versus testing time, we conducted several simulations for each model. Each diagram shows the mean of the simulation results as the line marked "m"; the confidence intervals above the standard deviation as the line marked "m+s"; and the confidence intervals below the standard deviation as the line marked "m-s." The standard deviation along the timeline is presented as the line marked by "s" at the bottom. These simulations neither validate nor invalidate whether a particular model fits an actual project's data, but merely show how easily the characteristics of such a process can be comparatively analyzed.

**Poisson-process simulation.** The NHPP is also easily simulated when the hazard function  $\lambda(t_a, t_b)$  is known in closed form. The program for counting the overall number of NHPP events that will occur over a given time interval is

```
#define produce(x) \
    random_poisson(x)
...
events = produce(lambda
    (ta, tb));
```

where `random_poisson(x)` is a subprogram that produces a Poisson-distributed random value when passed the parameter `x`. Donald Knuth has published an algorithm for generating Poisson random numbers.<sup>13</sup>

The time profile of an NHPP may be simulated by slicing the  $(0, t)$  interval into  $dt$  time slots, recording the behavior in each slot, and progressively accumulating the details to obtain the overall event-count profile, as in the following algorithm:

```
t = 0.;
while (t < t_max)
{ n = produce(lambda
    (t, t + dt));
```

```
/* n is the fine
   structure */
events += n;
t += dt;
}
```

The form of the cumulative rate function `lambda(t, t + dt)` may be extended to include a dependence on `events`, thereby causing the algorithm above to approximate a nonhomogeneous Markoff event-count process with increasing fidelity as `dt` becomes sufficiently small that multiple events per `dt` interval become rare. As mentioned previously, however, the behavior of such simulations may be indistinguishable, even at larger `dt`, on the basis of single realizations of the event process. This hybrid form can speed up the simulation by removing the necessity of slicing time into extremely small intervals.

This modified form of the simulation algorithm is called the piecewise-Poisson approximation of the Markoff event-count process.

**Multiple categories of events.** If the set of events  $\{e_i: i = 1, \dots, n\}$  that were classed together previously are now partitioned into categorized subsets according to some given differentiation criteria — for example, faults distinguished as being *critical*, *major*, or *minor* — then the partitioning of events into categories likewise partitions their rate functions into corresponding categories, to which integers could be used as indices.

When an event occurs, the algorithm shown in the box on page 78 produces the index of a rate function. Finding this index among the categorized subsets of integers relates the event to the distinguished category of occurrences. You can thus easily simulate the behavior of multiple categories of events by changing from a single event counter, `events`, to an array of event counters, `events[ ]`, and altering the program as follows:

```
i = event_index(n, t);
c = event_category(n, i);
events[c]++;
```

The overall event-classification scheme is thus encapsulated within a single `event_category()` function for the entire categorization of events.

**Other event processes.** Often in the software life cycle, if an event of one type occurs there is a uniform probability  $p < 1$  that another event of a different type will be triggered. For example, suppose that for each unit of

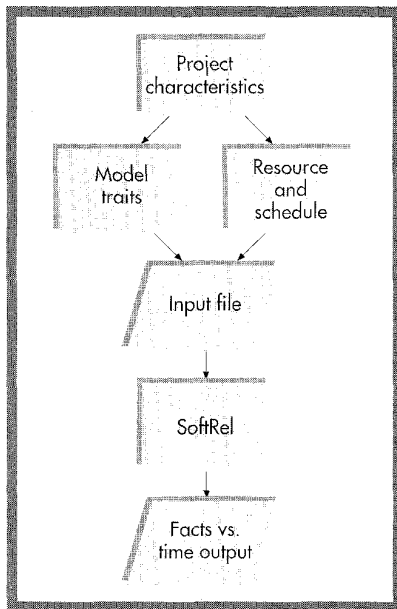
**Our method can simulate all event-rate random processes.**

code generated there is a probability  $p$  that a fault is created. If there are  $n$  events of the first type, then the  $k$  events of the second type are governed by the binomial distribution function, which is also easily simulated.<sup>14</sup>

Moreover, when  $n$  itself is a Poisson random variable with parameter  $\lambda$ , the distribution of  $k$  is also Poisson, with parameter  $p\lambda$ . Thus, occurrences of events of the second type may be simulated without actually counting events of the first type by using the `produce()` function with parameter  $p\lambda$ .

```
#define select(n, p) \
    random_binomial(n, p)
...
n = produce(lambda(t, t + dt));
k = select(n, p);
```

Finally, when there is an ultimate number of events  $N$  that a Poisson process may reach before it is terminated, and  $N$  is specified in advance, then the growth of events over time must be stopped after the  $N$ th occurrence. This type of *goal-limited process* is also easily simulated.



**Figure 2.** *SoftRel execution context. The two data sets of project attributes and scheduled resources are merged into a single file for processing by SoftRel, which outputs a file that can be imported into a spreadsheet for further analysis.*

**General event-rate processes.** The simulation method we describe here is more general than required for production of Markoff processes and NHPPs. The Poisson-process simulation algorithm just described springs directly from our method, which can simulate all event-rate random processes.

Thus we can simulate life-cycle activities that may have event-count dependencies between nonoverlapping time intervals as well as rate functions that depend on variable schedules and other irregularities over time. Whenever event functions produce homogeneous Markoff processes in a piecewise fashion, the event processes simulated during each of these segments will follow the piecewise-Poisson approximation. The programs presented previously can thus simulate a much more general and realistic reliability process than has been hypothesized by any other analytic model known to us.

The six programs described previously typify the methods traditionally used to analyze stochastic processes over a variety of input conditions. From a programming perspective,

then, we require very little sophistication to simulate a reliability process. Insight, care, and validation are required, however, in modeling the intricate system-dynamic interrelationships among the various rate functions that characterize that process.

### SOFTREL

We have embodied these simulation techniques in a reliability-process simulation package, SoftRel. It simulates the entire reliability life cycle, including the effects of interrelationships among activities. For example, SoftRel provides for an increased likelihood of faults injected into code as the result of missing or defective requirements specifications. SoftRel also acknowledges that testing requires the preparation and consumption of test cases, and that repairs must follow identification and isolation. SoftRel further requires that human and computer resources be scheduled for all activities.

The SoftRel package is a prototype, currently configured to simulate processes having constant event rates per causal unit. We do not advocate that constant-rate processes necessarily model software reliability, nor do we endorse the prototype as a model ready for industrial use. Rather, we regard it as a framework for experimentation, for generating data typical of analytic-model assumptions for comparison with actual collected project data, and for inference of project characteristics from comparisons. Other event-rate functions will be accommodated in later versions by changing current constant rates and other parameters to properly defined functions indicated by project histories.

The current input to SoftRel consists of a single file that specifies the  $dt$  time slice, about 70 traits of the software project and its reliability process,

and a list of activity, schedule, and resource allocations. Internally, these form a data structure called the `model`. Also internally, the set of status monitors at any given time are stored in a data structure called `facts`, which records

- ◆ the elapsed wall-clock time,
- ◆ the time and resources consumed by each activity — 42 measures in total, and
- ◆ a snapshot of 48 measures of project status.

SoftRel outputs a single file that contains the series of facts produced at each  $dt$  interval of time. SoftRel simulates two types of failure events: defects in specification documents and faults in code. Figure 2 shows the execution context of SoftRel. A project's characteristics are divided into two contexts:

- ◆ a fixed number of project attributes as embodied in numeric size and rate parameters, and
- ◆ a variable number of scheduled resources to be applied, each designating the event to which it applies, the time slot over which it is valid, the staff (work resource per unit time), and computer resources (CPU hours per unit time) available.

Both of these data sets are merged into a single file that forms the input `model` processed by SoftRel. The calculated response to the `model` is collected into a `facts` file and output by the program in a form suitable for input to a spreadsheet for plotting and further analysis.

**Major components.** SoftRel is initialized by setting sizes of items for construction, integration, and inspection. These could have been designed just to equal the goal values given in the model, but the model values are only approximate. Sizes are set to Poisson random values, with the model input values as means.

In a typical software-engineering life

cycle, several interrelated software-reliability subprocesses take place concurrently. The simulator uses 14 major components to characterize the activities in these subprocesses, with appropriate staffing and resource levels devoted to each activity:

1. *Document construction*: The simulator assumes that document generation and integration are piecewise-Poisson approximations with constant mean rates per workday specified in the model, not to exceed the goal values. Defects are assumed injected at a constant probability per documentation unit. At each injection of a defect, the document hazard increases according to the defect-detection characteristic.

2. *Document integration*: Document integration consists of acquiring reusable documentation, deleting unwanted portions, adding new material, and making minor changes. The simulator assumes that each of these subactivities is a goal-limited piecewise-Poisson approximation similar to the document-construction process. Each subactivity results in defect creation. Documentation is integrated at a constant mean rate per workday, and defects are injected at a constant probability per documentation unit. Hazard increases at each defect according to the defect-detection characteristic assumed. The total current documentation units consist of new units, reused minus deleted units, and added units; changes do not alter the total volume of documentation.

3. *Document inspection*: Document inspection is a goal-limited, piecewise-Poisson approximation of a type similar to document construction. Both new and integrated reused documentation are assumed to be inspected at the same rate and with the same efficiency. Documentation is inspected at a mean constant rate per workday. Inspected units are allocated among new documents and reused documents in pro-

portion to the relative amounts of documentation in these two categories. Defects detected during inspections may not exceed those injected; the simulator characterizes the discovery of defects as a goal-limited binomial process. The defect-discovery rate is assumed to be proportional to the current accumulated document hazard and the inspection efficiency.

4. *Document correction*: Defect corrections are produced at a rate determined by the staff level and the attempted-fix rate given in the model. Actual corrections take place according to the defect-fix adequacy, not to exceed the actual number of defects discovered — a goal-limited, binomial situation. Attempted fixes can also inject new defects and can change the overall amount of documentation. True corrections decrease the document hazard; the injection of new defects increases it.

5. *Code construction*: Code production follows the same formula as document construction. However, the average pace at which faults are created is influenced not only by the usual fault density that may occur as a normal consequence of coding, but also by the density of undiscovered defects in documentation and by the amount of missing documentation. Each fault injected increases the code hazard. However, whereas document defects are found only by inspection, code faults may be found by both inspection and testing, and at different rates.

6. *Code integration*: Simulation of code integration is similar to that for document integration, except that code units replace document units and coding rates replace documentation rates. The fault-injection rate is of the same form as that for code construction. Each fault increases the code hazard.

7. *Code inspection*: Code inspection mirrors document inspection. The number of faults discovered will not

exceed the total number of as yet undiscovered faults. The simulator assumes that the fault-discovery rate is proportional to the current accumulated fault hazard and the inspection efficiency. Because previously discovered faults may not yet have been removed

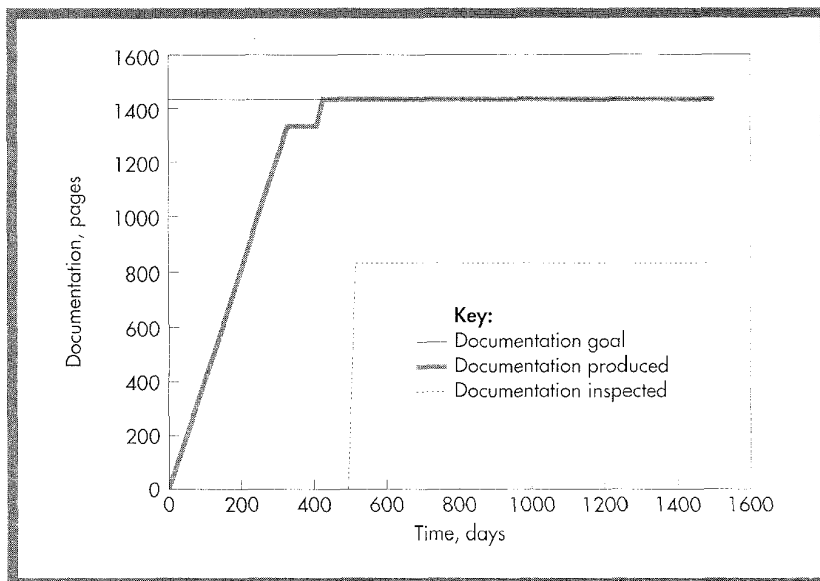
<b>Code faults may be found by both inspection and testing, and at different rates.</b>
---

at the time of discovery, the number of newly discovered faults is assumed to be proportional to the number of as yet undiscovered faults.

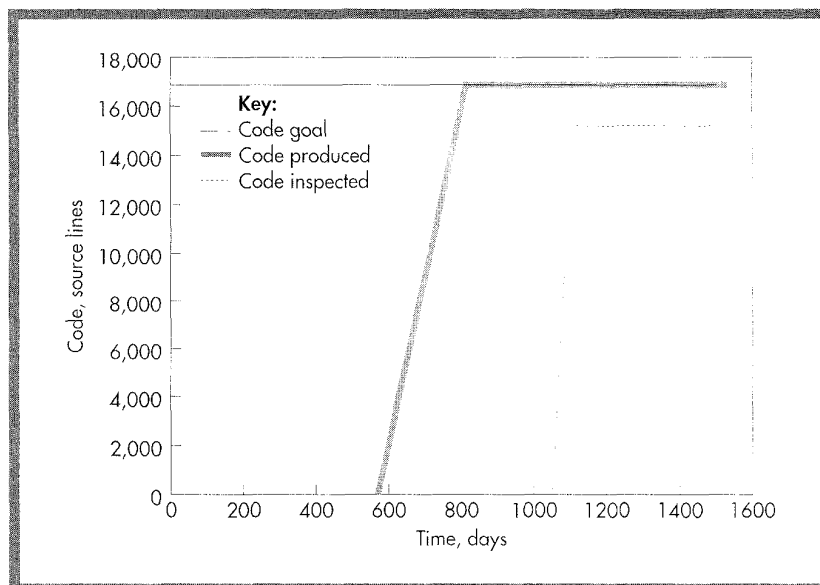
8. *Code correction*: Code-correction simulation follows the same algorithm given for document correction, translated to code units. Fault hazard is reduced upon correction of a fault and increased if any new faults are injected by the correction process. Documentation changes are produced at assumed constant mean rates per attempted correction.

9. *Test preparation*: Test preparation consists of producing a number of test cases in each *dt* slot in proportion to the test-preparation rate, which is a constant mean number of test cases per workday.

10. *Testing*: Testing simulation has two parts: If a test outage is in effect, the outage-time indicator decrements and the time-and-effort indicator increments; if an outage is not in effect, failures occur at the modeled rate — the number observed is computed as a binomial process regulated by the probability of observation. The failure rate function returns a value proportional to the current hazard level. The function also consumes computer resources and test cases, the latter at a mean constant rate.



**Figure 3.** Simulated document construction, integration, and inspection for the CDS project. Although the volume of documentation units reached its goal, because of inadequate resource allocation only about 63 percent of the documentation was actually inspected.



**Figure 4.** Simulated code construction, integration, and inspection for the CDS project. The volume of code units reached its goal and 90 percent of the inspection goal was met as well.

11. *Fault identification:* The total number of failures analyzed may not exceed the number of failures observed. Failures are analyzed at a mean constant rate per workday. The identification of faults is limited in number to those still remaining in the system. The isolation process is regulated by the fraction of faults remain-

ing undiscovered, the adequacy of the analysis process, and the probability of faithful isolation.

12. *Fault repair:* The number of attempted repairs may not exceed the number of faults identified by inspections and testing, less those corrected after inspection, plus those identified for rework by validation and retesting.

Of those attempted, a select number will really be repaired, while the rest will mistakenly be reported as repaired. Repairs are assumed here to be made on faults identified for rework first. A select number of new faults may be created by the attempt, and code units may be altered: deleted, added, or changed. Attempted repairs take place at a mean constant rate per workday.

13. *Validation of repairs:* The validation of attempted repairs takes place at an assumed mean constant rate per workday. The number of repairs validated may not exceed the number of repairs attempted. The number of faulty repairs detected is a select number determined by the probability that validation will recognize an unrepaired fault when one exists and the probability that unrepaired faults are among those attempted repairs being validated (the repair adequacy); the detected bad fixes cannot exceed the actual number of misrepaired faults. Detected bad fixes are designated for rework and removed from the unrepaired, undiscovered fault count.

14. *Retesting:* Retesting takes place at a mean constant number of retests per workday and consumes computer resources at the scheduled rate per day. No new test cases are generated or consumed, because the original test cases are assumed available for regression. Retesting is assumed to encounter only those failures caused by unrepaired faults.

**Input and output.** SoftRel tracks 70 input model parameters and 90 output facts parameters, all of which are described fully elsewhere.<sup>4</sup> The input file additionally contains a list of staffing and computer-resource packets, each of which allocates resources to specified activities and time slots. Time slots may overlap or leave gaps, at the discretion of the user. Such schedules are the natural outcome of

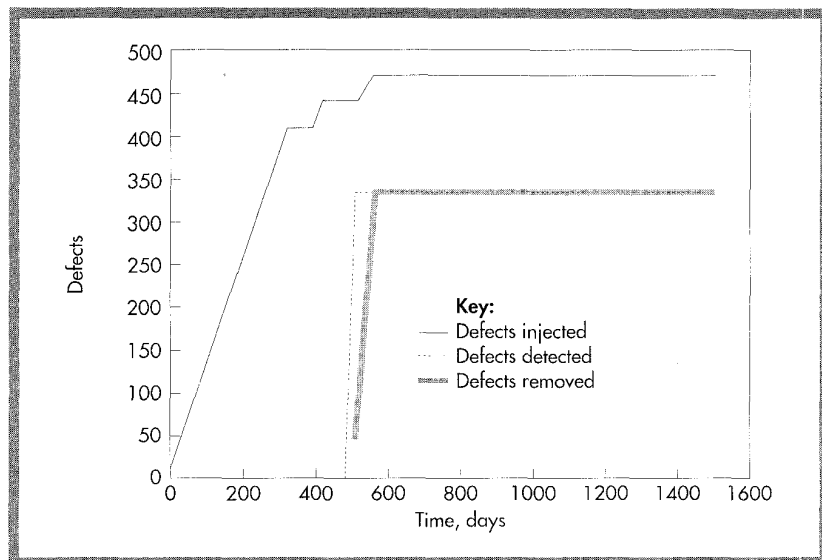


development-process planning and are of fundamental importance in shaping the reliability process. You need at least 14 schedule packets to allocate resources and time slots to each of the 14 assumed reliability process activities. More packets may appear when an activity repeats or has a nonconstant resource-allocation profile.

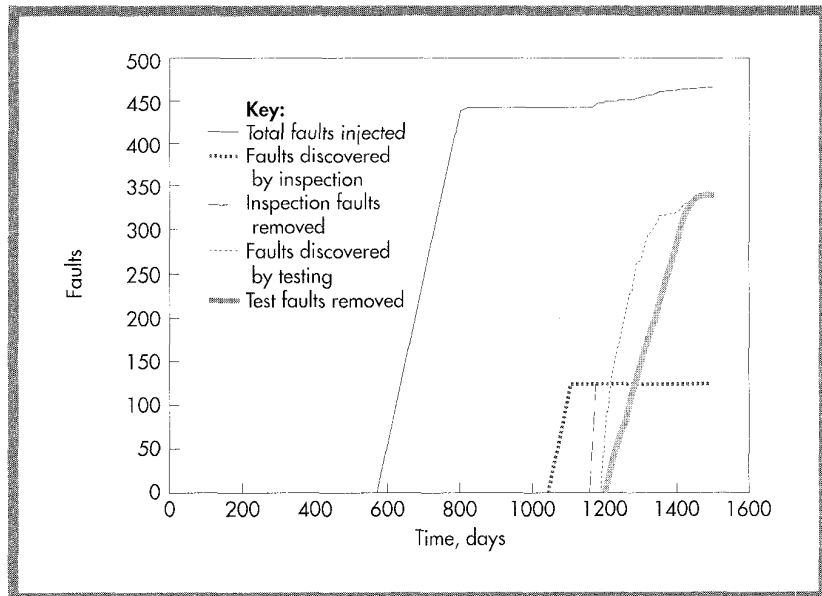
Output values consist of all product, work, CPU, resource, fault, failure, and outage values. These are time-tagged in the form of a `facts` data structure and written to the output file at each `dt` time interval for later scrutiny — plotting, trending, and `model` readjustments, for example — by other application programs such as spreadsheets.

We intend that the reliability process embodied in the prototype be fairly comprehensive with respect to what really transpires during software development. The simulator therefore requires parameters relating to the ways in which people and processes interact. The large number of parameters in the simulator might, at first, seem to present an overwhelming, impractical barrier to modeling, but you must remember that the true reliability process is even more complex. We felt that the number of parameters used was the least that would be capable of reproducing the realism we hoped for. Reducing the number of parameters might either reduce the fidelity of the simulation or the generality of the reliability-process model. Our assessment may change after sufficient experimentation has taken place, whereupon selective alterations or combinations of the parameters may be indicated. In any case, these parameters could be independently estimated and continuously refined with use.

If projects do not have sufficient data about past projects to give values to certain parameters, then sensitivity analyses using SoftRel can indicate which are the most influential and



**Figure 5.** Simulated defect discovery and correction for the CDS project. All the detected defects were corrected, but a sizable number of defects were inserted during the correction period and more than 100 defects were left in the documents.



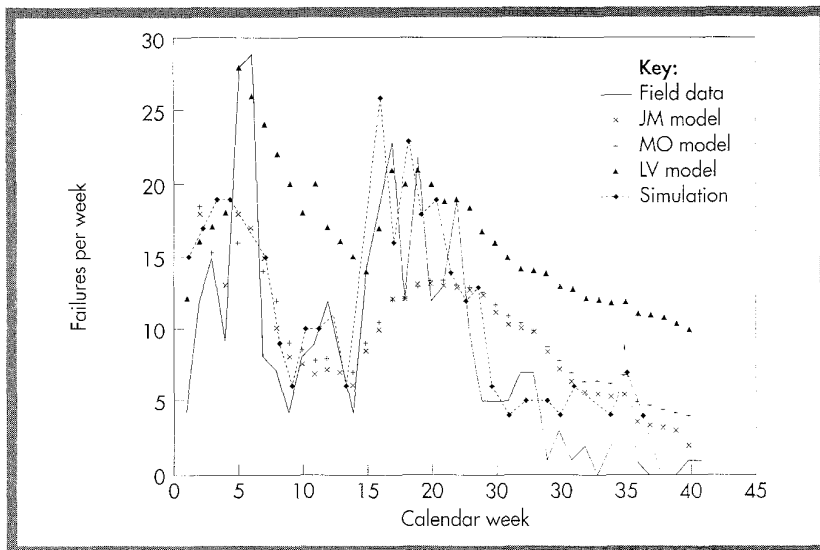
**Figure 6.** Simulated fault injection, discovery, correction, and repair for the CDS project. By the end of the project, about seven faults per KLOC had been found in inspections and corrected, and about 22 faults per KLOC had been uncovered by testing and removed; the fault density at delivery was about 0.2 faults per KLOC.

thereby in what area a metrics effort may prove most useful in reliability management. Alternatively, by making some of the parameters inactive, users may simplify the model to focus on only one or two activities at a time. This may be done by assigning typical or default values to the parameters — usually 0 or 1 — thereby reducing the

number of measured parameters to only those deemed pertinent and realistic within the project context.

### SOFTREL CASE STUDY

SoftRel has already been applied to a real-world project: a subsystem of the



**Figure 7.** Comparative predictions of failures per week for the CDS project. The SofiRel simulation technique produced a very good forecast that could have been used for tracking the reliability status during the entire testing phase: the error deviance for the simulation results is 24.5. As a comparison with the analytical software-reliability model results, the error deviances for the Jelinski-Moranda, Musa-Okumoto, and Littlewood-Verrall models are 38.8, 43.3, and 99.7, respectively.

Galileo Flight Project at the Jet Propulsion Laboratory. Here we describe that project, apply the simulation technique, and compare the results with those obtained from several traditional reliability models.

**Project description.** Galileo is an outer planet spacecraft project that began in 1977, a mission that was originally entitled "Jupiter Orbiter and Probe," or JOP. Unlike previous outer solar system missions, the Galileo orbiter was designed to remain in Jovian orbit for an extended period. This would allow observations of variations in planetary and satellite features over time, augmenting the information obtained by previous fly-by missions. Galileo was launched in October 1989 and reached the Jovian system in late 1995.

The Galileo spacecraft has two major on-board flight computers, largely embodied in software: The Attitude and Articulation Control Subsystem, and the Command and Data System. Our case study focuses on the CDS software-reliability profile.

The CDS flight software is real-time embedded software, written in 17,000 lines of assembly code (including 500 reused lines), with about 1,400

pages of documentation (including 100 reused pages), produced over a period of approximately 1,500 calendar days, excluding weekends. This project went through several design reviews and code inspections, underwent structured analysis and design, and recorded and tracked failures during its testing phase.

**Estimations and results.** When simulating an end-to-end development project based on data from the Galileo CDS project, we took some of the project parameters from project records, personnel within the project estimated other values, and we chose the remaining values as probably typical of this project's behavior despite the lack of immediately available data for them. For example, we adopted parameter values we believed to be typical of injecting faults in the correction and repair processes. None of the model input parameters was set to zero.

Thus, even though few verifiable model parameters were available outside the testing phase, we were able nevertheless to form an entire plausible hypothetical model to illustrate simulation of an end-to-end reliability process. Lacking better development life-cycle data, we presumed all CDS

activities other than testing — construction, inspection, and anomaly removal — took place serially, merely to observe what their simulated behaviors would be. To view typical Markoff reliability behavior, this overall study also presumed that each activity took place without resource and schedule variations.

Figures 3 through 6 show the simulated documentation, code, defect, and fault profiles of the software, sampled every 10 days. Of particular note are the behaviors of the documentation, code, injected defects, and injected faults — precisely those activities for which no project data exists. Because the numbers of units are comparatively large, the relative irregularity levels are low, as predicted from Equation 2.

Figure 3 shows that the volume of documentation units did reach its goal, but in this case, only about 63 percent of the documentation was actually inspected, even though the model placed a goal of 95 percent on inspection. This is an instance where inadequate resources were allocated to the inspection process: More resources would have been required to reach the goal. The effects of correcting defects on page count are not visible. The second rise in documentation is due to the integration of the reused 100 pages.

Figure 4 similarly shows that the volume of code units did reach its goal and that the 90 percent inspection goal was met as well. The effects of correcting and repairing faults on code size, however, are again not visible.

The injection, detection, and removal of defects, shown in Figure 5, are a little noisier than documentation and code production, but not much. All the detected defects were corrected, but a sizable number of defects were inserted during the correction period, which spanned days 520 to 580. Finally, more than 100 defects were left in the documents.

**TABLE 1**  
**CDS TESTING SCHEDULE**

Activity	Accumulated Failures	Begin Week	End Week	Full-Time Testers	CPU Usage
Functional test	90	0	5	2.0	0.4
Feature test	150	5	13	2.0	0.4
Operational test 1	300	13	23	2.0	1.2
Operational test 2	325	23	33	2.0	1.0
Operational test 3	341	33	40	2.0	2.0

Figure 6 shows the fault activity, which exhibits the noisiest behavior of all, but is still fairly regular. The initial rise in injected faults resulted from construction; the second rise, which is not visible, resulted from integration; the third, a sharp rise again, resulted from the imperfect fault-correction process; and the final, gradual rise resulted from the imperfect fault-repair process. By the end of the 1,500-day project, about seven faults per thousand lines of code had been found in inspections and corrected, and about 22 faults per KLOC had been uncovered by testing and removed. The fault density at delivery was about 0.2 faults per KLOC.

Although we consider the final fault-discovery count to be accurate, the time profile of the simulation results does not appear to be as irregular as the actual project data. It seems likely, then, that the fault-discovery process here is probably not homogeneous, either. On the basis of this case study, it appears that the simulation of all reliability subprocesses will require the use of nonhomogeneous event-rate models that reflect irregular workloads and schedules of life-cycle activities.

**Comparisons with other models.** To simulate the details of Galileo CDS testing activity, we separated its testing phase into five subactivities that had constant staffing but irregular CPU and schedule allocations, as Table 1 shows. These schedule parameters were obtained as those necessary to fit the simulator output to project data. The fit appears adequate to describe the underlying nature of the random-failure process.

Figure 7 shows the field data, the

results obtained from the piecewise-homogeneous simulation process, and the results from three other models: Jelinski-Moranda, Musa-Okumoto, and Littlewood-Verrall. For better visibility of process granularity, data is shown in the form of failures per week, rather than cumulatively. The JM, MO, and LV statistics were calculated to be "one-week-ahead" predictions, in which all the failure data up to a given week were used to predict the number of failures for the next week.

Figure 7 shows that SoftRel's simulation technique produced a very good forecast that could have been used for tracking the reliability status during the entire testing phase. The *error deviance* for the SoftRel simulation results in Figure 7 is 24.5, while the error deviances for the JM, MO, and LV models are 38.8, 43.3, and 99.7, respectively. We conjecture that the reliability forecast could have been accurately simulated prior to the start of testing, had actual schedule and resource plans been used *a priori*. The other models above were inadequate to predict even one week ahead; the LV model turned out to be particularly optimistic.

**R**eliability modelers seldom have the luxury of several realizations of the same failure process to test their hypotheses concerning the nature of a system's reliability. Nor are they ever provided with data that faithfully match the assumed natures of their models. Nor are they able to probe into the underlying error and failure mechanisms in a controlled way. Rather, they are faced with the problem of not only guessing the forms and particulars of

the underlying, random error and failure processes from the scant, uncertain data they possess, but also with the problem of best forecasting future failures from this single data set.

The assumptions of the SoftRel simulation approach are certainly less restrictive than those underlying analytic models. The simulation approach solves software-reliability prediction problems by producing data conforming precisely to the software-failure assumptions. Simulation enables investigation of questions such as, "How does a project's observed data compare with that emanating from an NHPP having the following characteristics?" and "Which analytic prediction model is the best under the following assumptions?" We believe that the SoftRel tool and its offspring will offer significant potential to researchers and practitioners in answering such questions, in evaluating the sensitivity of predictions to various error and failure modeling assumptions, and in forecasting software-project status profiles, such as time-lines of work products and the progress of testing, fault isolation, repair, validation, and retest efforts.

Simulation of a real-world project reinforced our confidence in the validity of the approach. We believe that homogeneous Markoff event-count models that uniformly consume resources do not adequately model the statistical failure profile of an actual project. The nonhomogeneous, variable-resource-schedule event-rate simulation model produced good early forecasts of reliability growth that could prove useful for process-status assessment.



We expect that further collaborations between government agencies and industry will continue to refine the reliability simulation technique and lead to a better understanding of the reliability process and to improvements in the SoftRel genre of tools. ●

## ACKNOWLEDGMENTS

We thank Yi-Bing Lin of National Chiao Tung University and Yu-Yun Ho of Bellcore for their valuable input. Portions of the research described in this paper were carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

## REFERENCES

1. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability - Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
2. M.R. Lyu and A. Nikora, "Using Software Reliability Models More Effectively," *IEEE Software*, July 1992, pp. 43-52.
3. A. von Mayrhauser et al., "On the Need for Simulation for Better Characterization of Software Reliability," *Proc. 4th Int'l Symp. Software Reliability Engineering*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 264-273.
4. R. Tausworthe, "A General Software Reliability Process Simulation Technique," Tech. Report 91-7, Jet Propulsion Laboratory, Pasadena, Calif., 1991.
5. *Handbook of Software Reliability Engineering*, M.R. Lyu, ed., McGraw-Hill and IEEE CS Press, New York, 1996.
6. W. Kreutzer, *System Simulation: Programming Styles and Languages*, Addison-Wesley, Reading, Mass., 1986.
7. Z. Jelinski and P.B. Moranda, "Software Reliability Research," in *Statistical Computer Performance Evaluation*, E.W. Freiberger, ed., Academic Press, San Diego, Calif., 1972, pp. 465-484.
8. A.L. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Trans. Reliability*, 1979, pp. 206-211.
9. J.D. Musa and K. Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *Proc. Seventh Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1984, pp. 230-238.
10. J.T. Duane, "Learning Curve Approach to Reliability Monitoring," *IEEE Trans. Aerospace*, 1964, pp. 563-566.
11. B. Littlewood and J.L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," *J. Royal Statistics Society*, 1973, pp. 332-346.
12. S. Yamada, M. Ohba, and S. Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection," *IEEE Trans. Reliability*, 1983, pp. 475-478.
13. D.E. Knuth, *The Art of Computer Programming: Semi-Numerical Algorithms*, Addison-Wesley, Reading, Mass., 1970.
14. N. Roberts et al., *Introduction to Computer Simulation*, Addison-Wesley, Reading, Mass., 1983.
15. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 1965.



### Software Reliability Engineering Handbook

by Michael R. Lyu

Contains the best current software reliability engineering practices that can be applied to a variety of real-world projects. It is a definitive guide to today's most-used reliability techniques, models, tools, and solutions. This book/disk set covers measurement, prediction, and the effects of product process metrics on operational software behavior. It also describes how to apply this knowledge in specifying and guiding software development, acquisition, use, and maintenance.

850 pages. April 1996.  
ISBN 0-07-039400-8.  
Catalog # RS00030  
\$66.50 Members / \$69.50 List

**To order call: +1-800-CS-BOOKS**



**Robert C. Tausworthe** is a senior research engineer and the chief technologist of the Information Systems Development and Operations Division of NASA's Jet Propulsion Laboratory. Previously, he served as software chief engineer of Deep Space Network Digital Systems, manager of JPL's Institutional Software Standards development, and as deputy software manager of the Galileo project. He is the author of *Standardized Development of Computer Software* (Prentice-Hall), 25 papers in software methodology and analysis, and more than 100 papers in communications theory and mathematics.

Tausworthe received a BSEE from New Mexico State University and an MSFE and PhD from the California Institute of Technology. He is a fellow of the IEEE and a member of ACM and Sigma Xi.



**Michael R. Lyu** is a member of the technical staff at AT&T Research Laboratories. Previously, he worked as a member of the technical staff at the Jet Propulsion Laboratory, as an assistant professor in the electrical and computer engineering department at the University of Iowa, and as a member of the technical staff in the Applied Research Area of Bellcore. Lyu initiated the first International Symposium on Software Reliability Engineering in 1990. He has edited two books, *Software Fault Tolerance* (Wiley) and *Handbook of Software Reliability Engineering* (IEEE and McGraw-Hill) and has published more than 50 papers in the areas of software-reliability engineering, software-fault tolerance, and distributed processing.

Lyu received a BS in electrical engineering from National Taiwan University, an MS degree in electrical and computer engineering from UC Santa Barbara, and a PhD in computer science from UCLA.

Address questions about this article to Tausworthe at Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, CA 91109; Robert.C.Tausworthe@jpl.nasa.gov; or Lyu at AT&T Research Labs, 600 Mountain Ave., Murray Hill, NJ 07974; lyu@research.att.com.