IN-61-CR

p-57

JPL Publication 91-7

A General Software Reliability Process Simulation Technique

Robert C. Tausworthe

April 1, 1991

NASA

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL Publication 91-7

A General Software Reliability Process Simulation Technique

Robert C. Tausworthe

April 1, 1991

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Acknowledgements

The author would like to acknowledge and thank Dr. Michael Lyu of the University of Iowa and Dr. Brian Beckman of the Jet Propulsion Laboratory for their encouragement, review, and suggestions in the preparation of this publication.

Abstract

This publication describes the structure and rationale of the generalized software reliability process, together with the design and implementation of a computer program that will simulate this process. Given assumed parameters of a particular project, the users of this program are able to generate simulated status timelines of work products; numbers of injected anomalies; and the progress of testing, fault isolation, repair, validation, and retest. Such timelines are useful, in comparison with actual timeline data, for validating the project input parameters, and for providing data for researchers in reliability prediction modeling.

Contents

List of Figures

Appendix

1 INTRODUCTION

Reliability is probably the most important quality attribute of systems taken as a whole, although safety, availability, and usability may also rank very high on the priority list. *Reliability* is defined as the probability that a system or entity will perform as required for a specified period of time under a given usage. This basic definition applies to all systems, subsystems, and individual units, including those which contain, or consist solely of, software. An instance of the system or entity not preforming as required is termed a *failure*. Although hardware and software failure mechanisms are totally different, there are many analogies among their characteristics, and many of the same mathematical techniques for characterization apply.

Reliability is not directly measurable, but must be inferred through model-based evidence. Quantities that are related to reliability and that are often used as quality indicators include [1]

- the instantaneous rate of failure, called the rate of occurrence of failure (ROCOF).
- the time-to-failure (TTF), described by a probability distribution, or by the mean of this distribution (MTTF).
- the number of faults manifest, described by a distribution, or expected value, or density per unit product.

Software developers produce documents, or specifications, and source code as work products. Developers sometimes make *errors*, or mistakes, that lead to defects and faults in these products. In this publication, we shall use the term *defect* to mean the result of an error made in a document, and the term *fault* to mean the result of an error made in the code. We shall use the term *anomaly* to refer generally to either. Anomalies manifest themselves as departures from requirements observed as failures during executions or as items of concern during inspections. We shall refer to the removals of anomalies of inspections as *corrections*, and the removals of faults found in testing as *repairs*.

Documentation usually defines acceptable performance of the code. But when an error is made during the expression of a requirement, defective performance may be specified, which will later be observed as a usability problem, provided that the code is a faithful implementation of its specifications. Clearly, the reliability process definition must encompass requirement defects, as well as design, coding, and ancillary anomalies.

The numbers of anomalies in software products, as well as the nature of those anomalies, are matters of fact and not of probability; yet, the possibility that anomalies will be encountered and observed under given circumstances within a stated time span is uncertain. Software reliability modeling is a set of techniques that applies probability theory and statistical analyses to assess the achieved reliability of work products, both quantitatively and objectively.

Software reliability quantification has been the subject of wide study since its modeling by Jelinski and Moranda [2] in 1972. Since that time, at least 40 different models [3] have been proposed. See the books by Musa [4] and Shooman [5] and the article by Abdel-Ghaly, et al. [6] for comparisons.

The primary focus of these studies is on methods that assess current reliability and forecast future operability, based on rational assumptions and observable failure data. None of the models extends over the entire reliability process; most tend to focus only on failure observance during testing or operations. Moreover, none of the forecasting models has emerged as "the best" predictor in all cases. This may be due to a number of factors, such as oversimplification of the failure process, the quality of observed data, the lack of sufficient data to make sound inferences, and/or serious differences between the proposed model and the underlying true error process(es). It is conceivable that the nature of the underlying failure process(es) may differ among individual software developments.

The usual assumptions for reliability modeling are:

1. The effects of all anomalies are independent.
2. The times between failures are much greater than the instruction cycle of the computer.
3. Test input is randomly selected.
4. The test space "covers" the use space.
5. All failures are observed when they occur.
6. Faults are immediately removed upon failure, or not counted again.

This publication attempts to characterize the *entire* reliability process, including the development of new documentation and the integration of existing documentation into the set of specifications and manuals that form the final document product; the construction of new code and the integration of reusable modules that form the final operational product; the inspection, testing, and failure observance processes; and finally, the anomaly isolation and removal processes. The composite process is implemented as a Monte Carlo simulation tool, $\mathcal{S}$oft$^{\mathrm{r}}$el[1], in which development project parameters and sub-process rate functions may be altered to accommodate various particular modeling hypotheses. A much more restricted simulation was reported by Levendel [7].

In particular, the last three reliability modeling assumptions above are not directly incorporated into the technique presented here. Whether the test space actually matches the use space is rarely known. We certainly *hope* that the test designers have done a proper job, but we cannot be sure. Simulation can mimic the testing method, but it cannot extrapolate what will happen in operations

[1]©1990, California Institute of Technology, Pasadena, CA.

afterward (a subsequent version could, however, simulate the test vs. usage coverage). Simulation can imitate a random error in detecting a failure when one has occurred, as well as any system outage that may result due to an observed failure. Furthermore, simulation can track those faults which have been removed, and those which have not, so multiple failures due to the same fault can be readily simulated. Thus, the set of assumptions about the reliability process is much less restricted for simulation than for the "usual" models.

The simulation approach avoids difficulties in obtaining closed-form solutions to intricate multi-activity reliability process problems, obviates the need to oversimplify the process model merely in order to obtain simplified statistics (such as mean behavior), and reveals the truer stochastic nature of the reliability process. Realizations (i.e., sample functions) of the process are *not* smooth, as the mean process behavior is apt to be. Monte Carlo simulations display the inherent variability of the process from one realization to the next. Observers of the simulations will better understand that fitting a given assumed model to data points obtained from a single observed process may perhaps not produce realizations typical of the given data.

The first purpose of $\mathcal{S}\mathrm{oft}^{\mathrm{r}}\mathrm{el}$ is experimental: *not* to provide reliability assessments directly, but to supply data for studying the merits of the various existing predictive models. Reliability modelers seldom have the luxury of several realizations of the same failure process. Nor are they provided with data that faithfully match the metrics 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 form and particulars of the underlying error and failure random processes from the scant, uncertain data they possess, but also with the problem of best forecasting future failures from that single data set. The simulator solves the latter problem when given a hypothesized answer to the former. That is, $\mathcal{S}\mathrm{oft}^{\mathrm{r}}\mathrm{el}$ enables the investigation of such questions as, How typical are the observed data to that emanating from a non-homogeneous Poisson process with the following characteristics? and Which of the following prediction methods is the best under the following assumptions? $\mathcal{S}\mathrm{oft}^{\mathrm{r}}\mathrm{el}$ is thus a tool that can assist in evaluating sensitivities of predictions to various error and failure modeling assumptions.

The second use of $\mathcal{S}\mathrm{oft}^{\mathrm{r}}\mathrm{el}$ is predictive. Once the entire reliability process and its parameters have been simulated, including variable manpower and scheduling of activities, trade-offs can be made to optimize resource allocations for construction, inspection, and testing, and to determine the merits and proper level of inspections both during and after construction of documents and code.

The simulation technique presented here does not necessarily assume stationary or homogeneous event statistics, nor that activities must occur in any particular order, nor that activities are entirely completed in any given time interval. Manpower and resources can be applied to arbitrary activities in an arbitrary number of arbitrarily scheduled time slots.

$\mathcal{S}\mathrm{oft}^{\mathrm{r}}\mathrm{el}$ recognizes that the lack of documentation and its defect content at any given time are related to the number of faults being injected into the code. If users have measured such effects in past projects, they may use this information to optimize resource and schedule planning. The construction and integration of reusable documents and code are characterized only to the extent that, as document and code units accumulate at specified rates, defects and faults become a part of the products at specified rates.

$\mathcal{S}$oft$^{\mathrm{r}}$el also allows simulation of imperfect correction and repair, and simulation of anomaly reinjection into products as a result of these activities. Additionally, since tests cannot be run unless test cases and computer resources are available, $\mathcal{S}$oft$^{\mathrm{r}}$el simulates the generation and application of test cases, and the corresponding consumption of resources.

2 DISCRETE EVENT PRELIMINARIES

The fundamental assumption of the reliability process simulation technique is that every stochastic event that occurs is the result of an underlying Markoff process [8]. A Markoff process is a time series of values $x(t)$ whose future statistics depend only on the current value of the process and a rate function, call it $\beta(t)$, where $\beta(t)\,dt$ for small enough dt acts as the conditional probability that the event occurs in the interval $(t,\, t+dt)$, given that it had not occurred before t.

If $\mathcal{O}$ and $\mathcal{E}$ denote the states of an event, $\mathcal{O}$ in effect before the event and $\mathcal{E}$ after its occurrence, then a particular member of the stochastic time series defined by $\{\beta(t), \mathcal{O}, \mathcal{E}\}$ is called a *sample function*, or *realization*, of the general discrete-event Markoff process.

The statistical behavior of this process is well-known: The probability that event $\mathcal{E}$ will have occurred prior to a given time t is related by the expression

$$\text{Prob}\{\mathcal{E} \text{ occurs in } (0,\, t)\} = P(\mathcal{E}, t) = 1 - \exp\left(-\int_0^t \beta(\tau)\,d\tau\right) \tag{1}$$

When the events of interest are failures, $\beta(t)$ is often referred to as the process *hazard function*.

If the integral of $\beta(t)$ is known in closed form, the event probability vs. time can be written down and analyzed. In all but the simplest cases, however, this analysis will require the aid of a computer. Even when the integral cannot be put in closed form, the integration can still be evaluated by using relatively simple, straightforward numerical analysis on a computer.

But if many related events are intricately combined in a problem, 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.

Alternatively, this single discrete-event process can be simulated rather easily, as illustrated by the computer algorithm in the following C language segment:

```
/*  dt is assumed set prior to this point  */

    event = 0;
    t = 0.;
    while (event == 0)
    {   t += dt;
        if (chance(beta(t) * dt))
            event++;
    }

/* the event has occurred at this point */
```

The dt in such simulations is always chosen such that the variations in $\beta(t)$ over the incremental time intervals $(t,\ t + dt)$ are negligible (for fidelity in instantaneous behavior), and such that $\beta(t)\,dt < 1$ (so that the instantaneous event probability is not unity). This type of discrete-event process simulation is called a ***time-driven event detection*** architecture.

Another approach to discrete-event simulation results in an ***event-driven*** process simulation architecture. In this scheme, the computer schedules the event for occurrence at some random later time, then waits until its virtual clock ticks out the appointed time, whereupon the event is made to occur. In its purest form, a simulator based on this architecture would compute the overall probability distribution of the event occurrence (see Eq. 1) by numerical integration for all values of t. Then it would pick the event time by using a random number generator having this distribution, and insert this time-tagged event into an event queue. However, if $\beta(t)$ is changing dynamically and stochastically due to occurrences of other interacting events, the pure event-driven simulation technique cannot be applied. In simulating the occurrences of several interacting events, only the next-occurring event can be scheduled at any given time deterministically. After this event occurs, the next subsequent event can then be scheduled, and so on. The complexity of this stochastic event-driven simulation thus becomes high. More information on event-driven process simulation may be found in [9].

Time-driven event simulation, on the other hand, does not schedule events, but allows them to "happen." It requires only examining t in dt-intervals using $\beta(t)$ up to the time of occurrence, rather than computing $P(\mathcal{E}, t)$ for all t. Moreover, the algorithm retains its simplicity, even when $\beta(t)$ is related dynamically and stochastically to other interacting events. It was for this reason that the time-driven event detection architecture was chosen for $\mathcal{S}$oft$^{\text{r}}$el.

Time-driven event detection simulation is a form of system dynamics simulation, with the distinction that the observables are discrete events randomly occurring in time. In systems dynamics simulations, the observables are typically non-stochastic and non-discrete. For more information on the systems dynamics approach, the reader may consult [10].

All examples given in this publication are illustrated in the C language. For brevity, and to focus on the core algorithms, declarations and incidental code are omitted. In the code segment above, **chance(x)** compares a $[0, 1)$-uniform **random()** value with **x**, thus attaining the specified conditional probability function. The **chance** function may be achieved by a macro definition,

```
double random(void);
#define chance(x) (random() < x)
```

2.1 Counting Events

If we do not stop the iteration in the previous algorithm when the event occurs, but instead continue to count the occurrences over a given interval $(0, t)$, the simulated event may now occur a random number of times. The algorithm for this simulation evolves to

```
/*  t and dt are assumed set prior to this point  */

    events = 0;
    x = 0.;
    while (x < t)
    {   x += dt;
        if (chance(beta(events, x) * dt))
            ++events;
    }

/* the event has occurred a number of times at this point */
```

In this example, the use of **beta(events, x)** acknowledges that the transition rate function may not only change over time, but also may be sensitive to the number of event occurrences up to the current time. Note that the counter, **events**, has been renamed in the plural to acknowledge that multiple occurrences are being counted.

Mathematically, we use $\beta_n(t)$ to denote that n occurrences of the event have occurred prior to t. When $\beta_n(t) = \beta_n$ is independent of time, the process is said to be ***homogeneous***; otherwise, it is *nonhomogeneous*. For a homogeneous process, β_n denotes that n occurrences of the event have occurred prior to starting the current process clock.

The probability $P_n(t)$ that n Markoff events occur in an interval $(0,\ t)$ is known to satisfy the Kolmogoroff equation [8]

$$\frac{P_n(t)}{dt} = -\beta_n(t)\,P_n(t) + \beta_{n-1}(t)\,P_{n-1}(t) \tag{2}$$

where $\beta_i(t) = 0$ and $P_i(t) = 0$ for $i < 0$. The solution takes the recursive form

$$\begin{aligned} P_0(t) &= e^{-\lambda_0(0,t)} && (3) \\ P_n(t) &= \int_0^t \beta_{n-1}(\tau)P_{n-1}(\tau)\,exp[\lambda_n(0,\tau) - \lambda_n(0,t)]\,dt && (4) \\ \lambda_n(t_0,t_1) &= \int_{t_0}^{t_1} \beta_n(\tau)\,d\tau && (5) \end{aligned}$$

For $n \geq 1$. In the homogeneous case, $\lambda_n(0,t) = \beta_n t$.

The probability that an event has not occurred prior to time t, namely, $P_0(t)$, above, agrees with Eq. 1. A general closed-form solution for $P_n(t)$ for $n > 0$, unfortunately, is unknown.

When the rate of occurrence is independent of the number of past occurrences, or when $\beta_n(t) = \beta(t)$ is independent of n, then it is well-known and straightforward to prove from the above formula that the number of times the event occurs is governed by the Poisson distribution,

$$P_n(t) = \text{Prob}\{\boldsymbol{n} = n\} = \frac{\lambda^n(0,t)}{n!} e^{-\lambda(0,t)} \tag{6}$$

$$\lambda(t_0, t_1) = \int_{t_0}^{t_1} \beta(\tau)\, d\tau \tag{7}$$

The time series of such events is understandably termed a *Poisson process*. In the homogeneous case, $\lambda(0,t) = \beta t$.

The statistics of Poisson occurrences in the interval $(t_0,\ t_1)$ are the same as over $(0, t_1 - t_0)$. In particular, the mean and variance of occurrences over a t-length interval are

$$n = E\{\boldsymbol{n}\} = \lambda(0,t) \tag{8}$$

$$\sigma_n^2 = \lambda(0,t) \tag{9}$$

$$\frac{\sigma_n}{n} = \frac{1}{\sqrt{\lambda(0,t)}} = \frac{1}{\sqrt{n}} \tag{10}$$

where $E\{\}$ represents the statistical expectation operator.

One may note from Eq. (10) that, as n increases, the percentage deviation of the process decreases; this signifies that Poisson processes involving large numbers of occurrences appear to become relatively regular. If physical processes appear more irregular than this, we shall not be able to simulate them using the Poisson form.

The Poisson process is easily simulated. The restrictions imposed earlier, namely, that $\beta_n(t)$ be nearly constant over the interval of observance and that $\beta_n(t)\, dt < 1$, no longer need be imposed when we are merely counting events over the time interval, provided we can compute $\lambda(0,t)$ and have a random number generator capable of producing Poisson-distributed samples.

Thus, if we are interested in the overall number of events that have occurred in a given time interval of a Poisson process, and not in the precise times of occurrence of each event, we can simplify the event-counting program merely to

```
#define produce(x)  random_poisson(x)
events  = produce(lambda(0, t));
```

where **random_poisson()** is a subprogram that returns a Poisson-distributed random value given the parameter **x**, and **lambda()** is the cumulative hazard. An algorithm for generating Poisson random numbers may be found in Knuth [11].

In this example, we have equated the term **produce** with **random_poisson** to emphasize the fact that a resource **lambda** is *producing* something, rather than that the number of **events** is merely a random number of a given type.

Other Markoff-event processes could conceivably be simulated using the technique above, but with **random_poisson** replaced by **random_number**. The problem with this formulation is that the equivalent cumulative hazard function **lambda()** and **random_number** generator are unknown, as noted above.

We may avert such problems, however, by concentrating on the fine structure of the event process, slicing the $(0, t)$ interval into dt time slots, observing the behavior in each slot, and progressively accumulating the details to obtain the overall event count profile. The independence of random variables in non-overlapping intervals of a Markoff process guarantees that this accumulated value possesses the same overall statistics as **events** formulated above. When the process does depend on the event history and is non-homogeneous, it is necessary merely to choose dt in the simulator so that

$$\int_t^{t+dt} \beta_n(t)\,dt \approx \beta_n(t)\,dt \tag{11}$$

and $\beta_n(t)\,dt < 1$. That is, we must consider intervals small enough that the process is homogeneous in each interval, with n inherited from one interval to the next.

```
t = 0.;
while (t < t_max)
{   n = produce(lambda(events, t, t + dt));
    events += n;
    . . .               /* n is the fine structure */
    t += dt;
}
```

Thus, even if a given Markoff process is not Poisson-distributed over long periods of time, it may yet be simulated by a set of Poisson processes juxtaposed in small enough adjacent non-overlapping time intervals. The time slice dt may be chosen small enough that the probable number of **++events** per slice is small, a condition fulfilled when $\lambda_n(t, t + dt)$ is small. If a time interval begins in state n (i.e., after the nth occurrence of $\mathcal{E}$), then $\beta_n(t)$ will not likely switch to $\beta_{n+1}(t)$ in the interval if the $\lambda_n(t, t + dt)$ is small. Processes that can pragmatically be represented thus are here called *Poisson interval* processes, or *piecewise-Poisson processes*.

For the remainder of this publication, we shall presume that processes of interest are of the piecewise-Poisson variety. We shall reflect on the implications of this assumption later, after looking at an example.

2.2 Multiple-Event Processes

Let us next consider the occurrences of several independent events, $\mathcal{E}_1, \ldots, \mathcal{E}_f$, with rate functions $\beta_n^{[1]}(t), \ldots, \beta_n^{[f]}(t)$, respectively, taken together as a class.

We may view the multi-event activity as if f algorithms of the single-event variety above were running simultaneously, each with its own separate rate function, **beta[i](n, t)**, controlling the nth occurrence of event $\mathcal{E}_i$ at time t. The probability that none of the events has occurred by time t is

$$P_0(t) = P_0^{[1]}(t) \cdots P_0^{[f]}(t) = \exp(-\sum_{i=1}^{f} \lambda_0^{[i]}(0,t)) = e^{-\lambda_0(0,t)} \tag{12}$$

with $\lambda_0(0,t) = \sum_{i=1}^{f} \lambda_k^{[i]}(0,t)$.

If each event is governed by a non-homogeneous Poisson process, then the first event occurrence statistics are of the non-homogeneous Poisson variety, with parameter $\lambda(0,t)$. Because of behavior-independence over non-overlapping time intervals, we may thus simulate the class-event occurrence process as a single piecewise-Poisson process governed by its composite rate function,

$$\lambda(t_0, t_1) = \sum_{i=1}^{f} \lambda_{n_i}^{[i]}(t_0, t_1) \tag{13}$$

When one of the events occurs in a very small interval $(t, t+dt)$, the probability that it was $\mathcal{E}_i$ is merely the relative occurrence rate

$$\text{Prob}\{\mathcal{E}_i | \mathcal{E} \text{ in } (t,\ t+dt)\} = \frac{\beta_{n_i}^{[i]}(t)}{\sum_{j=1}^{f} \beta_{n_j}^{[j]}(t)} \tag{14}$$

in which $n_k, k = 1 \ldots, f$ are the numbers of occurrences of the corresponding events prior to t. When given the fact that one of the events has occurred, we can simulate which one it was by using a $[0,1)$-uniform random number u to select that event $\mathcal{E}_i$ whose index i satisfies

$$\sum_{j=0}^{i-1} \beta_{n_j}^{[j]}(t) \le u\beta(t) < \sum_{j=1}^{i} \beta_{n_j}^{[j]}(t) \tag{15}$$

where $\beta_{n_0}^{[0]}(t) = 0$ and $\beta(t) = \sum_{j=1}^{n} \beta_{n_j}^{[j]}(t)$. This selection is expressed algorithmically by

```
int event_index(n, t)
{
    s = betasum(t);
    x = random() * s;
    i = n;
    while (x < s)
    {   s -= beta[i](events[i], t);
        i--;
    }
    return i;
}
```

When a new event $\mathcal{E}_i$ is added to the distinguished class of events, we merely readjust $\lambda(t_0, t_1)$ to include the corresponding $\lambda_{n_i}^{[i]}(t_0, t_1)$ function and proceed with the simulation. Similarly, when the class contracts to exclude an event $\mathcal{E}_i$, we decrease $\lambda(t_0, t_1)$ by $\lambda_{n_i}^{[i]}(t_0, t_1)$. This provides a simple and straightforward method of simulating the effects of injection and removal of defects and faults in computer software.

2.3 Multiple Categories of Events

Let the set of events $\{\mathcal{E}_i \ : \ i = 1, \ldots, n\}$ that were classed together above now be partitioned into categorized subsets according to some given differentiation criteria. For example, faults in a program could be distinguished as being either *critical*, *serious*, or *cosmetic*. The partitioning of events into categories likewise partitions their rate functions into corresponding categories, and equivalently, the bracketed indices of the rate functions into sets of integers.

The composite process remains of the piecewise-Poisson type; when an event occurs, the algorithm in the preceding Subsection produces the index of the rate function. Then, finding this index among the categorized index subsets relates the event to the distinguished category of occurrences. The one event counter, **events**, is replaced by an array of event counters, **events[ ]**, in the simple algorithm

```
    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 class of events.

2.4 Secondary Event Processes

Another type of event process of interest is the following: For each event of one type that occurs, there is a uniform probability $p < 1$ that another event of a different type is triggered. (For example, for each unit of code we generate, there is a probability p that we inject a fault.) If there are $\boldsymbol{n}$ events of the first type, then the $\boldsymbol{k}$ events of the second type are governed by the binomial distribution function

$$P(k|n) = \text{Prob}\{\boldsymbol{k} = k | \boldsymbol{n} = n\} = \binom{n}{k} (1-p)^{n-k} p^k \qquad (16)$$

When n itself is a Poisson random variable with parameter $\lambda(t_0, t_1)$, the distribution of k also turns out to be Poisson, with parameter $p\lambda(t_0, t_1)$. Thus, if we intend to simulate the occurrences of events of the second type without actually counting events of the first type, we may merely use the **produce()** function with parameter $p\lambda(t_0, t_1)$. However, if we wish to count events of the first type as well, we need to have a **random_binomial()** function too. (Knuth [11] also provides an implementation of this.) To emphasize here that a selection is taking place among produced events, we define a **select** equivalence.

```
#define select(n, p)  random_binomial(n, p)
```

The simulation code for the two processes now appears as

```
n = produce(lambda(t0, t1);
k = select(n, p);
```

The mean and variance of the selected numbers are

$$\bar{k} = E\{k\} = np \tag{17}$$

$$\sigma_k^2 = (1-p)pn = (1-p)\bar{k} \tag{18}$$

$$\frac{\sigma_k}{\bar{k}} = \sqrt{\frac{1-p}{\bar{k}}} < \frac{1}{\sqrt{\bar{k}}} \tag{19}$$

Whenever p is near unity or $\bar{k}$ is large, the mean percentage variation of the secondary process becomes small. When n derives from a Poisson process, its statistics will appear relatively regular when n is large, as discussed earlier. The secondary process may likewise appear very regular when k is large. The assumption of equiprobable selection in the simulation may thus need to be reconsidered in case the real-world physical process does not appear as smooth as the simulation.

A categorization of events, as for fault criticality, can also be made for rates of secondary events; certain categories of events can thus yield higher or lower rates of secondary event occurrences. The code changes only slightly, to

```
n = produce(lambda(0, t));
c = event_category(n, event_index(n, t);
k = select(n, p[c]);
```

2.5 Limited Growth Processes

When the final number of events N that a piecewise-Poisson process may reach before it is terminated is specified in advance, we permit the normal growth of **events** over time but stop the process after the Nth occurrence. This is easy if the dt time slots are very narrow, but **events** may at times otherwise overshoot N. To prevent overshooting, we define the **process** macro:

```
#define process(events, max, rate, increment)          \
{   if (max > events)                                  \
        increment = min(produce(rate), max - events); \
    else                                               \
        increment = 0;                                 \
    events += increment;                               \
}
```

The value of **events** cannot grow beyond **max** and the **increment** can be accessed, if fine structure is needed, by the rest of the program.

We similarly define the **find** macro as the limiting form of the **select** function, which chooses among **n** events with uniform probability **p**:

```
#define find(events, max, n, p, increment)            \
{   if (max > events)                                 \
        increment = min(select(n, p), max - events); \
    else                                              \
        increment = 0;                                \
    events += increment;                              \
}
```

PAGE______INTENTIONALLY BLANK

3 THE WORK ACTIVITY MODEL

Let us now focus on the various stochastic discrete-event-producing subprocesses that are taking place concurrently in the reliability process. Event occurrences in these subprocesses are interrelated and driven by the activities that are taking place. The events of importance are:

1. Composition, manifested by units of documentation and code produced. This may well include the reuse of existing units undergoing deletions, additions, and changes.
2. Human errors, which cause faults to be inserted in computer programs and defects into associated documents.
3. Discoveries of defects and faults by inspection.
4. Corrections, or attempted removals of identified defects and faults.
5. Generation of test cases, which are required later to seek out program faults and to validate corrections.
6. Failures, caused by execution of test cases that encounter faults.
7. Fault isolations, by which failures are made to correspond with the faults that caused them.
8. Repairs, or attempted removals of identified faults.
9. Validations, or desk checkings and reviews of repaired anomalies.
10. Retests, or regressions of test cases to assure that faults have been successfully repaired.

The activities of the life cycle during which these events occur are:

1. CONSTRUCTION: New documentation and code are generated while human errors inject faults and defects into them. Construction is divided into separate documentation and coding subactivities.
2. INTEGRATION: Reusable documentation and code are integrated with new documentation and code, while human errors inject defects and faults. Integration is divided into separate documentation and coding subactivities.
3. INSPECTION: Defects and faults are detected through examination of the software and documents. Inspections are also divided into document and code subactivities. Inspections may fail to recognize anomalies when encountered.

4. CORRECTION: Defects and faults are analyzed and corrected, in both document and code subactivities. Correction may be ineffective, and new anomalies may be injected.

5. PREPARATION: Test cases are generated.

6. TESTING: Test cases are executed and failures occur. Some failures may not be observed, and those that are observed may cause test outages.

7. IDENTIFICATION: Failure-to-fault correspondences are made. A particular isolation may assign the fault category as new, or previously seen, both correctly and erroneously.

8. REPAIR: Faults are removed (not necessarily perfectly) and, possibly, new human errors create new faults.

9. VALIDATION: Human checks affirm that repairs are effective, but may err in doing so; they may also detect that repairs are ineffective (i.e., faults were not removed), and may detect other remaining defects and faults.

10. RETEST: Re-execution of test cases verifies whether the repair is complete. If not, the repair is marked for re-repair. New test cases are assumed not needed. Retests may err in qualifying a fault as repaired.

These events and activities take place in scheduled time slots. For example, the REPAIR activity may be assigned to take place between times t_1 and t_2, with an assigned *staff* level, and an allocated *CPU* (or other) resource. A complete schedule is then a series of quintuples:

$$\begin{gathered}(activity,\ t_{begin},\ t_{end},\ staff,\ CPU)\\ \dots\\ (activity,\ t_{begin},\ t_{end},\ staff,\ CPU)\end{gathered}$$

that together define the overall reliability process. Slot times may overlap or leave gaps, at the discretion of the user. Such schedules are the natural outcomes of development process planning and are of fundamental importance in shaping the reliability process.

4 THE SIMULATION TECHNIQUE

A reliability process simulator should be able to respond to schedules and work plans and to report the performance of subprocesses under the plan. By viewing the simulated results, users may then replan, as necessary, to optimize. For this reason, the simulator described in this report does not assume staff, resources, or schedule models, but provides for quintuple inputs, as described in the previous section.

The simulator should also capture the effects of interrelationships among activities. For example, if documentation is missing, there is the added likelihood that errors will be introduced during the coding phase; if documentation contains defects, it is likely that they will translate into a number of faults in the code unless they are removed before the coding activity; testing cannot take place without test cases being prepared; repairs must follow identification and isolation; and so on.

The software reliability process simulator $\mathcal{S}$oft$^{\mathrm{r}}$el complies with the preliminaries in Section 2, conforms to the events and work activities of the preceding Section, and characterizes all events as piecewise-Poisson Markoff processes with explicitly defined event rate functions. The set of adjustable parameters input to $\mathcal{S}$oft$^{\mathrm{r}}$el is called the `model`; the set of event status monitors that describes the evolving process at any given time is called the set of `facts`. The `model` and `facts` are collected into data structures, which are described below.

The `model` and `facts` structures are defined so as to accommodate multiple categories of classes of events in the subprocesses of the overall reliability process, with each `model-facts` pair representing a separate class of events. Because of the usual assumption that event processes are independent, the same simulation technique could be applied simultaneously by using separate computer processors running the same algorithms for each class. If only a single processor were to be used, the same algorithms could be applied to each class separately, but interleaved in time, or else they could be run entirely separately. In entirely separate executions, the sets of results would be merged later into a proper time sequence.

For simplicity, in its initial form, the simulator reported here only accommodates a single category of events for each of the reliability subprocesses. Separate runs using different `model` parameters can be later merged to simulate performance of a single process that has multiple failure categories, if desired. Extension of $\mathcal{S}$oft$^{\mathrm{r}}$el to accommodate the more general case is not conceptually difficult, but has not yet been undertaken. Later versions may possibly include multiple failure categories, should this feature prove beneficial.

$\mathcal{S}$oft$^{\mathrm{r}}$el simulates two types of failure events, namely, defects in specification documents and faults in code, all considered to be in the same seriousness category, as reflected by the single set of `model` parameters. As an aside, we note that the seriousness category is often indicated by the probabilities of observation and outage, and the lengths of outages: a process with these quantities high will have highly visible and abortive failures, whereas when these probabilities are low, the process will have rarely noticed, inconsequential failures.

The "documentation" currently simulated by $\mathcal{S}$oft$^{\mathtt{r}}$el consists only of requirements, design, interface specifications, and other entities whose absence or defective nature can beget faults into subsequently produced code. Integration and test procedures, management plans, and other ancillary documentation, when deemed not to correlate directly with fault generation, are excluded. The presumption is that the likelihood of a fault at any given time increases proportionately to the amount of documentation missing or in error.

$\mathcal{S}$oft$^{\mathtt{r}}$el does not currently simulate the propagation of missing and defective requirements into missing and defective design and interface specifications; both requirements analysis and design activities are currently combined in the document construction and integration phases. All defects occur either in proportion to the amount of new and reused documentation, to the amount that was changed, deleted, and added, or to the number of defects that were reworked.

4.1 Event Status Monitors: Output

The event status indicators of interest, or **facts**, during the reliability process are the time-dependent values

DU	= Overall documentation units goal
DU_t	= Total number of documentation units built
DU_n	= New documentation units built
DU_r	= Acquired reused documentation units
DU_rd	= Reused documentation deleted units
DU_ra	= Reused documentation additional units
DU_rc	= Reused documentation changed units
E_d	= Human errors putting defects in all documentation
E_dn	= Human errors putting defects in new documentation
E_dr	= Human errors putting defects in reused documentation
DH	= Total documentation hazard, weighted defects
DH_n	= New documentation hazard, weighted defects
DH_r	= Reused documentation hazard, weighted defects
DI_t	= Inspected portion of all documentation, in units
DI_n	= Inspected portion of new documentation, in units
DI_r	= Inspected portion of reused documentation, in units
D	= Documentation defects detected
d	= Documentation defects corrected
CU	= Overall goal for code units
CU_t	= Total number of code units built
CU_n	= New code units built
CU_r	= Acquired reused code units
CU_rd	= Reused code deleted units

`CU_ra`	=	Reused code additional units
`CU_rc`	=	Reused code changed units
`E_f`	=	Human errors putting faults in all code
`E_fn`	=	Human errors putting faults in new code
`E_fr`	=	Human errors putting faults in reused code
`CH`	=	Total code hazard, weighted defects
`CH_n`	=	New code hazard, weighted defects
`CH_r`	=	Reused code hazard, weighted defects
`CI`	=	Inspected portion of all code, in units
`CI_n`	=	Inspected portion of new code, in units
`CI_r`	=	Inspected portion of reused code, in units
`e`	=	Code faults detected in inspection
`h`	=	Code faults corrected (healed)
`C`	=	Test Cases prepared
`c`	=	Test cases expended
`F`	=	Failures encountered during testing
`A`	=	Failures Analyzed for fault
`f`	=	Faults isolated by inspection and testing
`w`	=	Faults needing rework, revalidation, etc.
`u`	=	Faulty repairs
`R`	=	Fault repairs undertaken
`V`	=	Validations conducted of fault repairs
`RT`	=	Retests conducted
`r`	=	Faults actually repaired
`rr`	=	Faults re-repaired

"Documentation units" and "code units" are typically counted in pages of specifications and lines of source code, but other conventions are acceptable, provided that rate functions and parameters of the **model** are consistently defined.

Other status metrics **facts** of interest are

t	=	Current time.
T[i]	=	Cumulative time consumed by activity i.
W[i]	=	Cumulative work effort consumed by activity i.
cpu[i]	=	Cumulative CPU or other computer resource consumed by activity i.
outage	=	Total outage time due to failure.
active	=	Boolean indicator, true if the process has not yet terminated.

Note that the time-related activities above which measure times in *days* are expressed as *elapsed wall-clock time*. Conversions to effort in *workdays* and to CPU (or other) computer resource utilization in *resource-days* are **model**-related and addressed in the next Section.

4.2 The Simulation Input Parameters

The reliability process we have described is fairly comprehensive with respect to what really transpires during software development. The capability to mirror that process in a simulator will require a large number of parameters relating to the ways in which people and processes interact. If the number of parameters in the simulator seems overwhelming, remember that the true process is even more complicated; reducing the number of parameters can reduce the fidelity of the simulation. Uncertainty about the values of parameters reflects the lack of metrics in projects to date and our ignorance of the underlying physical process.

The $\mathcal{S}$oft$^{\rm r}$el `model` parameters are the following:

`dt`	= simulation time increment
`workday_fraction`	= average days worked per schedule day per individual
`doc_new_size`	= new documentation units
`doc_reuse_base`	= reused document units
`doc_reuse_deleted`	= documentation units deleted from reuse base
`doc_reuse_added`	= documentation units added to reuse base
`doc_reuse_changed`	= documentation units changed in reuse
`doc_build_rate`	= new documentation units per workday
`doc_reuse_acq_rate`	= reuse acquisition rate, documentation units per workday
`doc_reuse_del_rate`	= reused documentation deletion rate, units per workday
`doc_reuse_add_rate`	= reused documentation addition rate, units per workday
`doc_reuse_chg_rate`	= reused documentation change rate, units per workday
`defects_per_unit`	= defects generated per documentation unit
`reuse_defect_rate`	= initial defects per unit in reused documentation
`del_defect_rate`	= defects inserted per deleted unit
`add_defect_rate`	= defects inserted per added unit
`chg_defect_rate`	= defects inserted per changed unit
`hazard_per_defect`	= document hazard units added or removed per defect injected or corrected
`new_doc_inspect_frac`	= fraction of new documentation inspected
`reuse_doc_inspect_frac`	= fraction of reused documentation inspected
`insp_doc_units_per_workday`	= inspected documentation units per workday
`find_rate_per_defect`	= fraction of documentation defects detected per inspected unit
`defect_fix_rate`	= corrected documentation defects per workday

`defect_fix_adequacy`	= true documentation fixes per correction
`new_defects_per_fix`	= defects created per correction
`doc_del_per_defect`	= documentation units deleted per correction
`doc_add_per_defect`	= documentation units added per correction
`doc_chg_per_defect`	= documentation units changed per correction
`code_new_size`	= new code units
`code_reuse_base`	= reused code units
`code_reuse_deleted`	= reused code units deleted
`code_reuse_added`	= code units added to reuse base
`code_reuse_changed`	= code units changed in reuse base
`code_build_rate`	= new code build rate, code units per workday
`code_reuse_acq_rate`	= reused code acquisition rate, code units per workday
`code_reuse_del_rate`	= reused code deletion rate, code units per workday
`code_reuse_add_rate`	= reused code addition rate, code units per workday
`code_reuse_chg_rate`	= reused code change rate, code units per workday
`faults_per_unit`	= faults generated per code unit
`reuse_fault_rate`	= initial reuse faults per unit
`del_fault_rate`	= faults inserted per deleted unit
`add_fault_rate`	= faults inserted per added unit
`chg_fault_rate`	= faults inserted per changed unit
`faults_per_defect`	= faults inserted per document defect density
`miss_doc_fault_rate`	= faults inserted per code unit generated per missing documentation fraction
`hazard_per_fault`	= code hazard units added or removed per fault injected or repaired
`new_code_inspect_frac`	= fraction of new code inspected
`reuse_code_inspect_frac`	= fraction of reused code inspected
`insp_code_units_per_workday`	= inspected code units per workday
`find_rate_per_fault`	= fraction of faults detected per inspected unit
`fault_fix_rate`	= corrected code faults per workday
`fault_fix_adequacy`	= true fault fixes per correction
`new_faults_per_fix`	= faults created per correction
`code_del_per_fault`	= code units deleted per fault
`code_add_per_fault`	= code units added per fault
`code_chg_per_fault`	= code units changed per fault
`tests_gen_per_workday`	= test cases generated per workday
`tests_used_per_day`	= test cases consumed per computer resource unit

failure_rate_per_fault	= failures per resource day per fault density
miss_code_fail_rate	= failures per resource unit per missing code fraction
prob_observation	= fraction of failures actually observed
prob_outage	= fraction of failures causing testing outage
outage_time_per_failure	= outage delay after failure, days
analysis_rate	= failures analyzed per workday
analysis_adequacy	= faults recognized per failure analyzed
repair_rate	= attempted fault repairs per workday
repair_adequacy	= true repairs per attempted repair
new_faults_per_repair	= faults created per attempted repair
validation_rate	= repairs validated per workday
find_rate_per_fix	= detected bad repairs per validation per unrepaired fault
retest_rate	= retested validated faults per workday
retest_adequacy	= detected bad repairs per retest per unrepaired fault
schedule	= pointer to **schedule item** packets

When the work effort expended by an activity is needed, it may be computed by using the instantaneous ***staffing***, or ***work force***, function $s(\alpha, t)$ defined for each such activity α over the time periods of applicability. The corresponding work effort $w(\alpha, T)$ over a time interval $(0, T)$, for example, is

$$w(\alpha, T) = \int_0^T s(\alpha, t)\, dt \tag{20}$$

In $\mathcal{S}$oft$^{\text{r}}$el, $s(\alpha, t)$ is coded as **staffing(A, p, M)**, where **A** is the activity, p points to a **facts** structure, and **M** points to a **model**.

Similarly, if computer CPU time, or another computer resource, is required in calculating the event-rate functions above, it is found through the conversion function $q(\alpha, t)$, which is defined for each activity α as the CPU or resource utilization per wall-clock day. The CPU resource usage over the time interval (0, T), for activity α, for example, is

$$T_{\text{cpu}}(\alpha, T) = \int_0^T q(\alpha, t)\, dt \tag{21}$$

The function $q(\alpha, t)$ in $\mathcal{S}$oft$^{\text{r}}$el appears as **resource(A, p, M)**, with the same arguments as **staffing**, above.

The number of wall-clock days may be interpreted either as literal calendar days, or as actual workdays. These alternatives are selected by proper designation of the **model** parameter, **workday_fraction**. A value of unity signifies that time and effort accumulate on the basis of one workday effort per schedule day per individual. A value of 5/7 means that work effort and resource utilization accumulate on the average only during 5 of the 7 days of the week. A value of 230/365 denotes that 230 actual workdays span 365 calendar days. These compensations are made in the **staffing** and **resource** functions, above.

Activities of the life cycle are controlled by the staffing function. No progress in an activity takes place unless it has an allocated work force. If, however, staffing is non-zero, event rates involve $s(\alpha, t)$ when work effort dependencies exist, and $q(\alpha, t)$ when CPU dependencies are manifest.

Staffing and computer resource allocations in the **model** are made via the **schedule** list of **schedule item** packets, each of which contains

activity	=	index of the work activity
t_begin	=	beginning time of the activity, days
t_end	=	ending time of the activity, days
staffing	=	staff level of the activity, persons
cpu	=	resources available, units per day
next	=	pointer to next **schedule item** packet

The entire list is merged by the staffing and resource-utilization functions, s and q, or **staffing** and **cpu** in the program, to provide scheduled workforce and computer resources at each instant of time throughout the process. Both **staffing** and **cpu** express resource units per project day. If the schedule quintuples include weekends, holidays, and vacations, then staff and resource values must be compensated so that the integrated staff and resources over the project schedule are the allocated total effort and resource values. This is done via the parameter **workday_fraction** discussed above.

4.3 The Simulation Algorithms

The extension of the algorithms in Section 2 to include all the events, activities, **model** parameters, and status **facts** is now fairly straightforward, as will be demonstrated. The **model** structure parameters are available through the pointer variable **M**, and the **facts** about a particular project are referenced via the pointer **p**. The overall program structure is

```
/* INITIALIZE  */
...
while (p->active)
{   p->t += M->dt;
    /* SET CURRENT STAFF AND RESOURCE LEVELS */
    ...
    /* DOCUMENT CONSTRUCTION  */
    ...
    /* DOCUMENT INTEGRATION   */
    ...
```

```
        /* DOCUMENT INSPECTION    */
        ...
        /* DOCUMENT CORRECTION    */
        ...
        /* CODE CONSTRUCTION      */
        ...
        /* CODE INTEGRATION       */
        ...
        /* CODE INSPECTION        */
        ...
        /* CODE CORRECTION        */
        ...
        /* TEST PREPARATION       */
        ...
        /* TESTING                */
        ...
        /* FAULT IDENTIFICATION   */
        ...
        /* FAULT REPAIR           */
        ...
        /* VALIDATION OF REPAIRS  */
        ...
        /* RETESTING              */
        ...
        /* DISPLAY OF RESULTS     */
        ...
    }
```

Each ellipsis (...) above indicates a body of code performing the indicated function. Each function is visited at each iteration; if staff is assigned, the function is performed and the status is recorded. Each will be described in turn.

4.3.1 Initialization

We begin by setting the event goals for construction, integration, and inspection. We could just equate the goal values to those given in the `model`, but the `model` values are probably only approximate anyway. Rather, we assume these events are produced by a piecewise-Poisson process, with the `model` values as means:

```
    DU_n  = produce(M->doc_new_size);
    DU_r  = produce(M->doc_reuse_base);
    DU_rd = produce(M->doc_reuse_deleted);
    DU_ra = produce(M->doc_reuse_added);
    DU_rc = produce(M->doc_reuse_changed);
    DU_rt = DU_r - DU_rd + DU_ra;
    p->DU = DU_n + DU_rt;
    DI_n  = M->new_doc_inspect_frac * DU_n;
```

```
DI_t  = DI_n + M->reuse_doc_inspect_frac * DU_rt;
new_doc_frac =  DI_n /  DI_t;

CU_n  = produce(M->code_new_size);
CU_r  = produce(M->code_reuse_base);
CU_rd = produce(M->code_reuse_deleted);
CU_ra = produce(M->code_reuse_added);
CU_rc = produce(M->code_reuse_changed);
CU_rt = CU_r - CU_rd + CU_ra;
p->CU = CU_n + CU_rt;
CI_n  = M->new_code_inspect_frac * CU_n;
CI_t  = CI_n + M->reuse_code_inspect_frac * CU_rt;
new_code_frac =  CI_n /  CI_t;
outage = 0.;
randomize(-1, 0);
```

The latter two statements reset the test outage indicator to 0 and seed the random number generator with a randomly chosen value. Note that the goal quantities bear the same identifiers as do their p-structure counterparts. This is a notational convenience, but may be confusing if the dereferencing "p->" is unnoticed.

4.3.2 Set Current Staffing and Resource Levels

The `staffing(A, p, M)` function serves two purposes: When invoked with `A` set to the SET_LEVELS activity designation, it computes and stores for later access all the staff and resource levels for the remainder of the activities at the current time. When called with `A` set to a particular activity designation, the staffing function returns the value of the current staff level devoted to that activity. The code in the main loop is merely

```
staffing(SET_LEVELS, p, M);
```

The staffing function computes the work force and resources by following the `M->schedule` chain of schedule quintuples input from the file of `model` parameters and adding together for each activity the quantities allocated in the current time slot. The `workday_fraction` parameter multiplies both staffing and resource levels to align the time and effort scales.

4.3.3 Document Construction

Document generation and integration are assumed to be piecewise-Poisson processes with constant mean rates per workday specified in the `model` not to exceed the goal values. Defects are injected at a constant probability per documentation unit. If later studies indicate that the documentation rate is nonhomogeneous, the `defects_per_unit` parameter will have to be changed into function form, as are all the rate functions in the current simulator. In order for the `select()`

function to be probabilistic, it is necessary that the **defects_per_unit** value be made less than unity by proper choice of documentation units. For example, if the documentation unit is chosen as the page, then there must be less than one defect per page, on the average. If there are more, then a smaller portion of a page should be chosen as the documentation unit.

At each injection of a defect, the document hazard increases according to the following defect detection characteristic, which is discussed further under Document Inspection.

```
process(p->DU_n, DU_n, rate(p, M, DOC_CONSTRUCTION, ANY), n);
if (n > 0)
    {   i = select(n, M->defects_per_unit);
        add_hazard(i, DOC, NEW, p, M);
    }
```

The **rate()** function supplies the document construction $\lambda(t_0, t_1)$ values for the given project status **p**, model **M**, and activity. The additional argument **ANY** is unused here, but selects among subactivities in integration phases. The **rate()** function also serves to accumulate time, work effort, and computer resource utilization totals across major activities.

4.3.4 Document Integration

Document integration consists of acquisition of reusable documentation, deletion of unwanted portions, addition of new material, and minor changes. Each of these subactivities is assumed to be a goal-limited piecewise-Poisson process of a type similar to the construction process described above. Defects are created as a result of each subactivity. 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.

```
process(p->DU_r, DU_r, rate(p, M, DOC_INTEGRATION, BASE), n);
i = 0;
if (n > 0)
    i += select(n, M->reuse_defect_rate);
process(p->DU_rd, DU_rd, rate(p, M, DOC_INTEGRATION,
    DELETION), n);
if (n > 0)
    i += select(n, M->del_defect_rate);
process(p->DU_ra, DU_ra, rate(p, M, DOC_INTEGRATION,
    ADDITION), n);
if (n > 0)
    i += select(n, M->add_defect_rate);
process(p->DU_rc, DU_rc, rate(p, M, DOC_INTEGRATION,
    CHANGE), n);
if (n > 0)
    i += select(n, M->chg_defect_rate);
    if (i > 0)
        add_hazard(i, DOC, BASE, p, M);
    p->DU_t = p->DU_n + p->DU_r + p->DU_ra - p->DU_rd;
    doc_frac = 1. - (double) p->DU_t / (double) p->DU;
```

The total current documentation units consist of new, reused minus deleted, and added units; changes are deemed not to alter the total volume of documentation. The document completion fraction **doc_frac** is later used to influence the fault-injection rate, a lower fault rate corresponding to a higher fraction of documentation completed.

4.3.5 Document Inspection

Document inspection is a goal-limited piecewise-Poisson process, 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 proportion to the relative amounts of documentation in these two categories.

Defects detected may not exceed those injected; the discovery of defects is characterized 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. Since previously discovered defects may not have been corrected at the time of rediscovery, the number of newly discovered defects is assumed to be proportional to the number of undiscovered defects.

```
process(p->DI_t, DI_t, rate(p, M, DOC_INSPECTION, ANY), n);
if (n > 0)
{   find(p->DI_n, DI_n, n, new_doc_frac, i);
    p->DI_r += n - i;
    find(p->D, p->E_d, n, inspect_eff(p, M, DOC) *
        new_defect_frac(p, M), n);
}
```

4.3.6 Document Correction

Defect corrections are produced at a rate determined by the staff level and 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 via the numbers of documentation units deleted, added, and changed. True corrections decrease the document hazard, while the injection of new defects increases it.

```
n = produce(rate(p, M, DOC_CORRECTION, ANY));
if (n > 0)
{   find(p->d, p->D, n, M->defect_fix_adequacy, i);
```

```
        remove_hazard(i, DOC, p, M);
        i = select(n, M->new_defects_per_fix);
        add_hazard(i, DOC, ANY, p, M);
        p->DU_rd += produce(i * M->doc_del_per_defect);
        p->DU_ra += produce(i * M->doc_add_per_defect);
        p->DU_rc += produce(i * M->doc_chg_per_defect);
    }
```

4.3.7 Code Construction

Production of code follows the same formulation as does document construction. However, the average pace at which faults are created is influenced not only by the usual number of `faults_per_unit` 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. This average pace must not exceed one fault per code unit. The code unit must be defined to make this possible. For example, if the code unit is chosen to be a line of code, we should not expect to see more than one fault per line of code, on the average.

Each fault injected increases the code hazard. But whereas document defects are only found by inspection, code faults may be found both by inspection and testing, and at different rates. When a certain hazard is injected during code construction, the means by which that fault will later be detected is unknown. The simulator reported here currently assumes that the hazard functions of faults found by inspection and by testing are related by a constant factor.

```
    process(p->CU_n, CU_n, rate(p, M, CODE_CONSTRUCTION, ANY), n);
    if (n > 0)
    {   pace = M->faults_per_unit
              + M->faults_per_defect * (p->E_d - p->D) / p->DU / n
              + M->miss_doc_fault_rate * doc_frac;
        i = select(n, pace);
        add_hazard(i, CODE, NEW, p, M);
    }
```

4.3.8 Code Integration

Simulation of code integration is comparable in structure to 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, above. The pace of fault injection must not exceed one fault per code unit. Each fault increases the code hazard.

```
    process(p->CU_r, CU_r, rate(p, M, CODE_INTEGRATION, BASE), n);
    i = 0;
    if (n > 0)
        i += select(n, M->reuse_fault_rate);
    process(p->CU_rd, CU_rd, rate(p, M, CODE_INTEGRATION,
        DELETION), n);
```

```
if (n > 0)
{   pace = M->del_fault_rate
        + M->faults_per_defect * p->E_dr / p->DU / n
        + M->miss_doc_fault_rate * doc_frac;
    i += select(n, pace);
}
process(p->CU_ra, CU_ra, rate(p, M, CODE_INTEGRATION,
    ADDITION), n);
if (n > 0)
{   pace = M->add_fault_rate
        + M->faults_per_defect * (p->E_d - p->D) / p->DU / n
        + M->miss_doc_fault_rate * doc_frac;
    i += select(n, pace);
}
process(p->CU_rc, CU_rc, rate(p, M, CODE_INTEGRATION,
    CHANGE), n);
if (n > 0)
{   pace = M->chg_fault_rate
        + M->faults_per_defect * (p->E_d - p->D) / p->DU /n
        + M->miss_doc_fault_rate * doc_frac;
    i += select(n, pace);
}
p->CU_t = p->CU_n + p->CU_r + p->CU_ra - p->CU_rd;
add_hazard(i, CODE, BASE, p, M);
```

4.3.9 Code Inspection

Code inspection mirrors the document inspection process, except that the number of faults discovered will not exceed the total number of as yet undiscovered faults. The fault discovery rate is assumed to be proportional to the current accumulated fault hazard and the inspection efficiency. Since previously discovered faults may not yet have been removed at the time of discovery, the number of newly discovered faults is assumed to be in proportion to the number of undiscovered faults.

```
process(p->CI_t, CI_t, rate(p, M, CODE_INSPECTION, ANY), n);
if (n > 0)
{   find(p->CI_n, CI_n, n, new_code_frac, i);
    p->CI_r += n - i;
    find(p->e, p->E_f - p->f, n, inspect_eff(p, M, CODE) *
        new_fault_frac(p, M), n);
}
```

4.3.10 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.

```
n = produce(rate(p, M, CODE_CORRECTION, ANY));
if (n > 0)
{   find(p->h, p->e, n, M->fault_fix_adequacy, i);
    remove_hazard(i, CODE, p, M);
    i = select(n, M->new_faults_per_fix);
    add_hazard(i, CODE, ANY, p, M);
    p->CU_rd += produce(n * M->code_del_per_fault);
    p->CU_ra += produce(n * M->code_add_per_fault);
    p->CU_rc += produce(n * M->code_chg_per_fault);
}
```

4.3.11 Test Preparation

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

```
p->C += produce(rate(p, M, TEST_PREPARATION, ANY));
```

4.3.12 Testing

The testing activity simulation has two parts: If a test outage is in effect, decrement the outage indicator and count the time and effort during the outage. If an outage is not in effect, produce failures at the modeled rate; the number observed is computed as a binomial process that is regulated by the probability of observation. Of those observed, a `select` number will cause outages with the given `model` probability of outage. The outage time per failure is assumed to be constant; this restriction can be lifted when more of the statistical nature of outages is learned.

The failure `rate` function returns a value proportional to the current hazard level. The function additionally consumes computer resources and test cases, the latter at a mean constant rate.

```
if (outage > 0.0)
{   outage -= (x = fmin(outage, M->dt));
    p->T[TESTING] += x;
    p->W[TESTING] += M->staffing(TESTING, p, M) * x;
}
else
{   p->F += (n = select(produce(rate(p, M, TESTING, ANY)),
            M->prob_observation));
    if (n > 0)
    {   outage = M->outage_time_per_failure *
                select(n, M->prob_outage);
        p->outage += outage;
        }
    }
```

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

```
process(p->A, p->F, rate(p, M, FAULT_ID, ANY), n);
if (n > 0)
find(p->f, p->E_f - p->h, n,
        new_fault_frac(p, M) * M->analysis_adequacy, i);
```

4.3.14 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. Changes to documentation, at an assumed constant mean rate per repair attempt, may be required.

```
process(p->R, p->e - p->h + p->f + p->w, rate(p, M, REPAIR, ANY), n);
if (n > 0)
{   p->r += (i = select(n, M->repair_adequacy));
    if (i > p->w)
    {   p->rr += p->w;
        p->w   = 0;
    }
    else
    {   p->rr += i;
        p->w  -= i;
    }
    p->u += n - i;
    remove_hazard(i, CODE, p, M);
    i = select(n, M->new_faults_per_repair);
    add_hazard(i, CODE, ANY, p, M);
    p->CU_rd += produce(n * M->code_del_per_fault);
    p->CU_ra += produce(n * M->code_add_per_fault);
    p->CU_rc += produce(n * M->code_chg_per_fault);
}
```

4.3.15 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 attempted repairs. 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 inadequacy); the detected bad fixes cannot exceed the actual number of mis-repaired faults. Those that are detected are scheduled for rework and removed from the unrepaired, undiscovered fault count.

```
process(p->V, p->R, rate(p, M, VALIDATION, ANY), n);
if (n > 0)
{   pace = M->find_rate_per_fix * (1. - M->repair_adequacy));
    i = minval((value) select(n, pace, p->u);
    p->w += i;
    p->u -= i;
}
```

4.3.16 Retesting

Retesting takes place at a mean constant number of retests per workday and consumes computer resources at the scheduled rate per day. No test cases are generated (or consumed), the assumption being that cases are already available for regression. Retesting is assumed to encounter only those failures due to unrepaired faults.

The number of retests may not exceed the number of validations. The number of bad fixes discovered by retesting will be a **select** number of the retests, which is regulated by the probability that a retest will discover the bad fix and the probability that a bad fix is among those attempted repairs retested. The latter probability is the probability that a bad fix was generated times the probability that validation missed the bad fix.

```
process(p->RT, p->V, rate(p, M, RETESTING, ANY), n);
if ( n > 0)
{   pace = 1. - M->repair_adequacy;
    pace *= M->retest_adequacy * (1 - M->find_rate_per_fix * x);
    i = minval((value) select(n, pace, p->u);
    p->w += i;
    p->u -= i;
}
```

4.4 Output and Display of Results

$\mathcal{S}$oft$^{\rm r}$el outputs to the standard device **stdout** and optionally to a file, when so named on the command line. When enabled, the first two records of the output file are strings bearing the name of the **model** file and the date the output was generated.

The display and writing of the status **facts** at each time increment is handled by the statement

```
show(p, M);
```

which writes the entire p structure to **stdout** and to the optional file, if selected. Writing thus occurs at each **dt** interval for display and use by other software programs.

The output file, when named on the command line, takes the form

model file name
date
*facts**

The first line in the file is a string giving the name of the **model** file that created the output and is terminated by a newline. The second line is the date that output was created, also terminated in a newline. The remainder of the file is made up of **facts** structures, one for each **dt** interval computed by the simulator. These are written in **struct** form,

```
facts *p;
...
fwrite(p, sizeof(facts), 1, p);
```

4.5 Preliminary Rate Functions

This Section presents the forms of the rate functions chosen for initial studies. The assumptions of the simulation are that all hazards are time-independent over short **dt** intervals and that event hazards are proportional to the numbers of event generators. For example, failure hazards are assumed to be proportional to the number of remaining faults.

The structure of the **rate** function is principally a switch that selects the rate characteristics for each given phase and subphase:

```
if ((staff = staffing(phase, p, M)) IS 0.0)
    return 0.0;

switch (phase)
{   case DOC_CONSTRUCTION:
        ...
    case DOC_INTEGRATION:
        switch (subphase)
        {   case BASE:
                ...
            case DELETION:
                ...
                case ADDITION:
                    ...
                case CHANGE:
                    ...
            }
            break;
        case DOC_INSPECTION:
            ...
            ...
            ...

        case TESTING:
            ...
        case FAULT_ID:
            ...
        case REPAIR:
            ...
        case VALIDATION:
            ...
        case RETESTING:
            ...
    }
    p->T[phase]    += dt;
    p->W[phase]    += staff * dt;
    p->cpu[phase] += resource(phase, p, M) * dt;
    return (pace * dt);
```

4.5.1 Document Construction Rate

The pace of constructing new documentation units is assumed to be proportional to the build-rate `model` parameter; however, the number of new units must not exceed the new-unit bound `DU_n`. The `work_fraction` function is used to reduce `dt` to a value that prevents this from occurring.

```
case DOC_CONSTRUCTION:
    pace = staff * M->doc_build_rate;
    dt *= work_fraction(pace, dt, DU_n - p->DU_n);
    break;
```

4.5.2 Document Integration Rate

Document integration takes place in four subphases: (1) a base activity, in which documentation for reuse is acquired, (2) a deletion activity, in which portions are deleted, (3) an addition activity, in which new portions are added to the existing packages, and (4) a change activity, in which documentation units are altered in a way that does not affect size. The `dt` interval is apportioned among each activity in the phase, according to the relative magnitudes of the rate functions and modified by the **work_fraction** function, so that the numbers of acquired, deleted, added, and changed units do not exceed their goal values.

```
case DOC_INTEGRATION:
    switch (subphase)
    {   case BASE:
            a = work_fraction(M->doc_reuse_acq_rate, dt,
                DU_r  - p->DU_r);
            b = work_fraction(M->doc_reuse_del_rate, dt,
                DU_rd - p->DU_rd);
            c = work_fraction(M->doc_reuse_add_rate, dt,
                DU_ra - p->DU_ra);
            d = work_fraction(M->doc_reuse_chg_rate, dt,
                DU_rc - p->DU_rc);
            if ((A = a + b + c + d) <= 0.)
                return 0.;

            dt *= a / A;
            pace = staff * M->doc_reuse_acq_rate;
            break;
        case DELETION:
            if (A <= 0.)
                return 0.;

            dt *= b / A;
            pace = staff * M->doc_reuse_del_rate;
            break;
        case ADDITION:
            if (A <= 0.)
                return 0.;

            dt *= c / A;
            pace = staff * M->doc_reuse_add_rate;
            break;
        case CHANGE:
            if (A <= 0.)
                return 0.;

            dt *= d / A;
            pace = staff * M->doc_reuse_chg_rate;
    }
    break;
```

4.5.3 Document Inspection Rate

Document inspections are assumed to examine a certain average number of units per day, with an inspection limit DI_t not to be exceeded.

```
case DOC_INSPECTION:
        pace = staff * M->insp_doc_units_per_workday;
        dt *= work_fraction(pace, dt, DI_t - p->DI_t);
        break;
```

4.5.4 Document Correction Rate

Document correction takes place at a fixed average rate, but the corrected defects may not exceed the number of defects so far discovered.

```
    case DOC_CORRECTION:
        pace = staff * M->defect_fix_rate;
        dt *= work_fraction(pace, dt, p->D - p->d);
        break;
```

4.5.5 Code Construction Rate

Like documentation, new code construction takes place at a constant mean rate, but may not exceed the target number of code units CU_n.

```
    case CODE_CONSTRUCTION:
        pace = staff * M->code_build_rate;
        dt *= work_fraction(pace, dt, CU_n - p->CU_n);
        break;
```

4.5.6 Code Integration Rate

Code integration simulation mirrors documentation in form, but differs in units, rates, and limits.

```
    case CODE_INTEGRATION:
        switch (subphase)
        {   case BASE:
                a = work_fraction(M->code_reuse_acq_rate, dt,
                    CU_r  - p->CU_r);
                b = work_fraction(M->code_reuse_del_rate, dt,
                    CU_rd - p->CU_rd);
                c = work_fraction(M->code_reuse_add_rate, dt,
                    CU_ra - p->CU_ra);
                d = work_fraction(M->code_reuse_chg_rate, dt,
                    CU_rc - p->CU_rc);
                if ((A = a + b + c + d) <= 0.)
                    return 0.;

                pace = staff * M->code_reuse_acq_rate;
                dt *= a / A;
                break;
            case DELETION:
                if (A <= 0.)
                    return 0.;
                pace = staff * M->code_reuse_del_rate;
                dt *= b / A;
                break;
            case ADDITION:
                if (A <= 0.)
                    return 0.;

                pace = staff * M->code_reuse_add_rate;
                dt *= c / A;
                break;
            case CHANGE:
                if (A <= 0.)
                    return 0.;

                pace = staff * M->code_reuse_chg_rate;
                dt *= d / A;
        }
        break;
```

4.5.7 Code Inspection Rate

As with documentation, code is assumed to be inspected at a constant mean rate, but may not exceed the allocated inspection limit **CI_t**.

```
    case CODE_INSPECTION:
        pace = staff * M->insp_code_units_per_workday;
        dt *= work_fraction(pace, dt, CI_t - p->CI_t);
        break;
```

4.5.8 Code Correction Rate

Code faults detected in inspections are assumed to be corrected at a constant mean fix rate, but the number **h** fixed (healed) may not exceed the number **e** found by inspections.

```
    case CODE_CORRECTION:
        pace = staff * M->fault_fix_rate;
        dt *= work_fraction(pace, dt, p->e - p->h);
        break;
```

4.5.9 Test Case Preparation Rate

Test cases are generated at a constant mean rate per applied workday.

```
    case TEST_PREPARATION:
        pace = staff * M->tests_gen_per_workday;
        break;
```

4.5.10 Test Failure Rate

If no code units exist, or if no CPU resource is available, testing cannot take place. If code units do exist and if testing can take place, the number of test cases that will be needed is determined from the case usage per CPU hour and the allotted CPU time. However, the number of test cases used may not exceed the number prepared above, less those that have been previously used in testing. (We assume that regression testing takes place in the retest phase, below; all new test cases are used in the testing phase.) When the unperformed tests are exhausted, testing ceases (`dt` is shortened). The number of test cases applied adds to the cumulative test case usage.

There are two contributors to failures: (1) the hazard injected per code unit and (2) the hazard that missing code will be encountered during a test. The former assumes a constant average failure rate per resource unit expended per fault per code unit (essentially a Jelinski-Moranda model using CPU time, rather than clock time, as in Musa [4]), and the latter assumes a constant average number of failures per resource unit expended per fraction of the total code so far produced.

```
case TESTING:
    if (p->CU_t IS 0 OR (cpu = resource(TESTING, p, M)) IS 0.0)
            return 0.0;

    cases = produce(case_use_rate(p, M) * cpu * dt);
    if (cases > (n =  (double)(p->C - p->c)))
    {       dt *= n / cases;
            cases = n;
    }
    if (cases <= 0.)
            return 0.;
    p->c += (value) cases;
    code_frac = 1. - p->CU_t / p->CU;
    pace = (M->failure_rate_per_fault * p->CH / p->CU +
       M->miss_code_fail_rate * code_frac) * cpu;
    break;
```

4.5.11 Fault Identification Rate

Failure analysis and fault identification take place at an assumed constant average rate, if there are failures to analyze; but the number `A` of failures that can be analyzed may not exceed the number `F` encountered so far during testing.

```
case FAULT_ID:
    pace = staff * M->analysis_rate;
    dt  *= work_fraction(pace, dt, p->F - p->A);
    break;
```

4.5.12 Fault Removal Rate

Fault repair, or removal, takes place at the same assumed constant average repair rate as the code correction activity, above. The number of repairs and corrections may not exceed the total number of faults discovered through testing and inspections.

```
case REPAIR:
    pace = staff * M->repair_rate;
    dt  *= work_fraction(pace, dt, p->e + p->f - p->r - p->h);
    break;
```

4.5.13 Repair Validation Rate

Validation of repairs is assumed to require a constant average effort per validation. However, the number `V` of validations may not exceed the number `R` of repairs.

```
case VALIDATION:
    pace = staff * M->validation_rate;
    dt  *= work_fraction(pace, dt, p->R - p->V);
    break;
```

4.5.14 Retesting Rate

Retesting takes place at a constant average number of retests per applied resource unit. The number `RT` of retests may not exceed the number `V` of repairs validated.

```
case RETESTING:
    pace = resource(RETESTING, p, M) * M->retest_rate;
    dt  *= work_fraction(pace, dt, p->V - p->RT);
    break;
```

[illegible] INTERNATIONALE [illegible]

5 EXAMPLE AND COMMENTARY

This Section presents the results of a simple hypothetical project to illustrate the simulated reliability process. The individual **model** parameters are listed in the Appendix. The salient characteristics of the project are that it will generate about 750 pages of documentation (500 pages new, 250 pages reused) and 15K lines of code (10K lines new and 5K lines reused) over a period of 450 days (interpreted as 90 weeks at 5 days per week). Some of the parameters have been obtained from historical data; other values are anecdotal, believed to be typical, but unsubstantiated; still others were arbitrarily chosen to illustrate a certain behavior believed to occur in projects, such as the injection of faults in the repair and correction processes, where the author had no readily available data. None of the **model** parameters is zero.

Figures 1 through 4 show the documentation, code, defect, and fault status during the project period. Of particular note is the smoothness of the rise of documentation and code during construction and integration. Because the numbers of units are comparatively large, the relative noise levels are low, as predicted from Eqs. 10 and 19. Although unsubstantiated by actual data as of this writing, it is doubtful that this seemingly linear behavior reflects actuality. The reason may be schedule irregularities: people generating documents and code do not exclusively dedicate their time to this activity every day of the applicable phases. They have other duties during these periods as well. If such effects were programmed into the **model** schedule fine structure, the construction and inspection processes could easily look more erose.

Figure 1 shows that the volume of documentation did reach its goal, but in this case, only about 70% of the documentation was actually inspected, even though the **model** placed a goal of 90% on inspection. More resources would have been required to reach the goal. The effects of correcting defects on page count are not visible.

Figure 2 similarly shows that the volume of code did reach its goal and that the 90% inspection goal was met as well. The effects of correcting and repairing faults on code size, however, are again not visible.

The injection and removal of defects, shown in Figure 3, are a little noisier, but not much. All the detected defects were corrected ($\texttt{D} = \texttt{d}$), but a sizable number of defects were inserted during the correction period (days 130-150). Finally, more than 150 defects were left in the documents.

The fault activity is shown in Figure 4. It exhibits the noisiest behavior of all, but is still fairly regular. The initial rise in injected faults is due to construction; the second rise due to integration; the third, a sharp rise again, is due to the imperfect fault correction process; and the final, gradual rise, is due to the imperfect fault repair process. By the end of the 450-day project, about 8 faults per kiloline of code had been found in inspections and corrected, and about 7 faults per kiloline of code had been uncovered by testing and removed; the fault density at delivery was about 7 errors per kiloline of code.

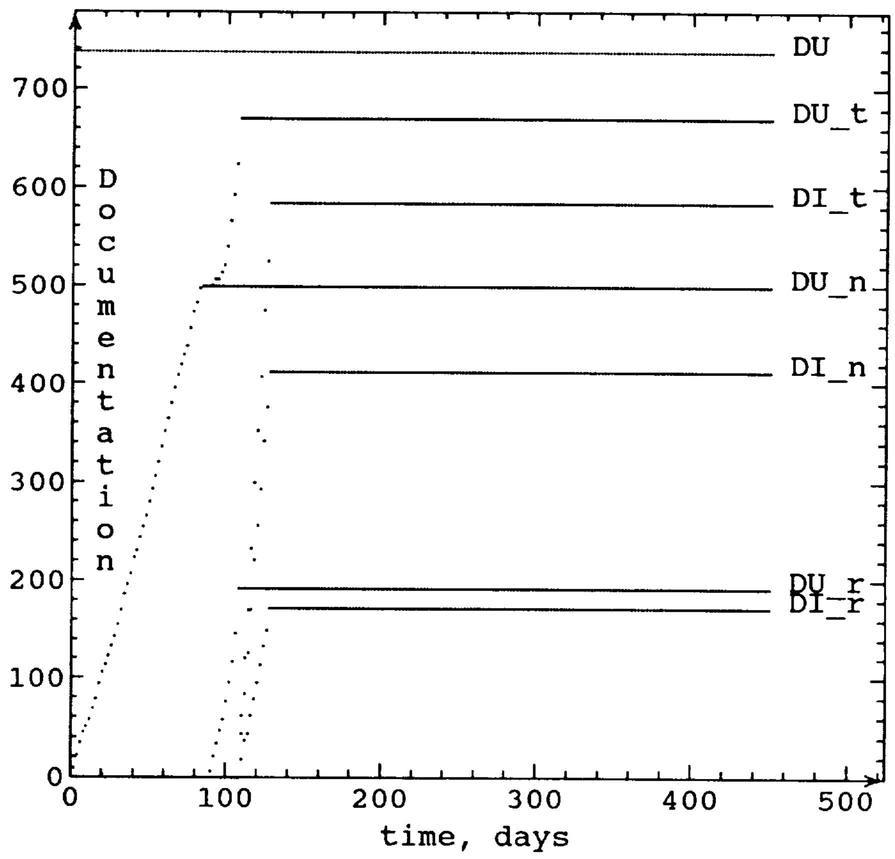

Figure 1: Document Construction, Integration, and Inspection.

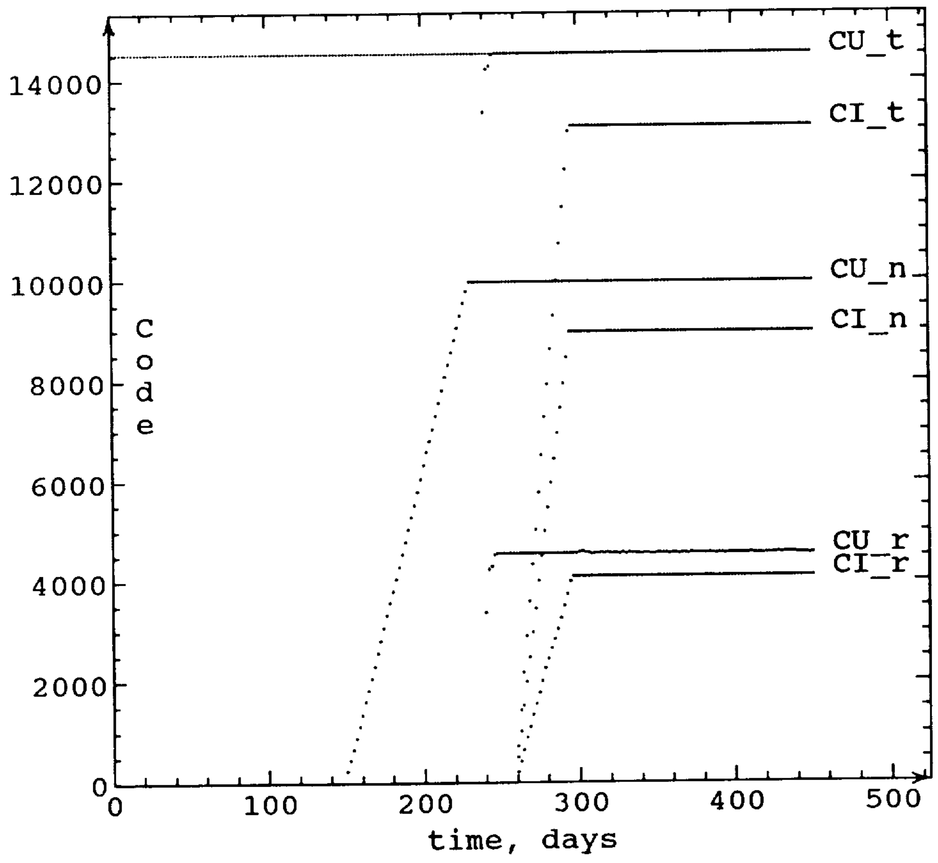

Figure 2: Code Construction, Integration, and Inspection.

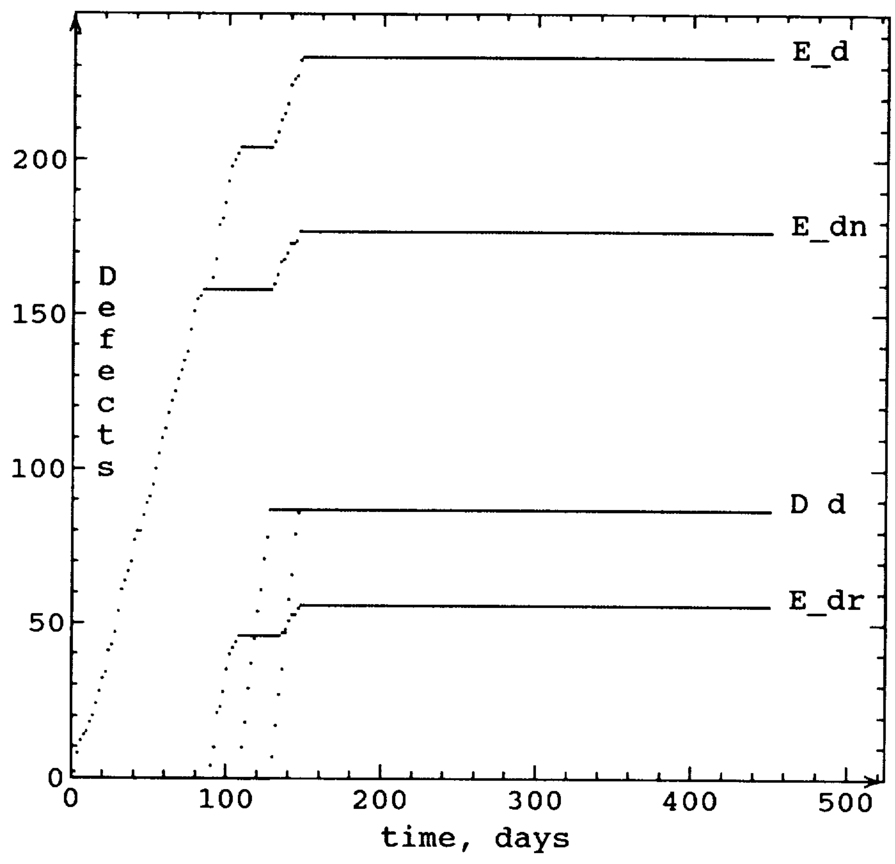

Figure 3: Defect Discovery and Correction.

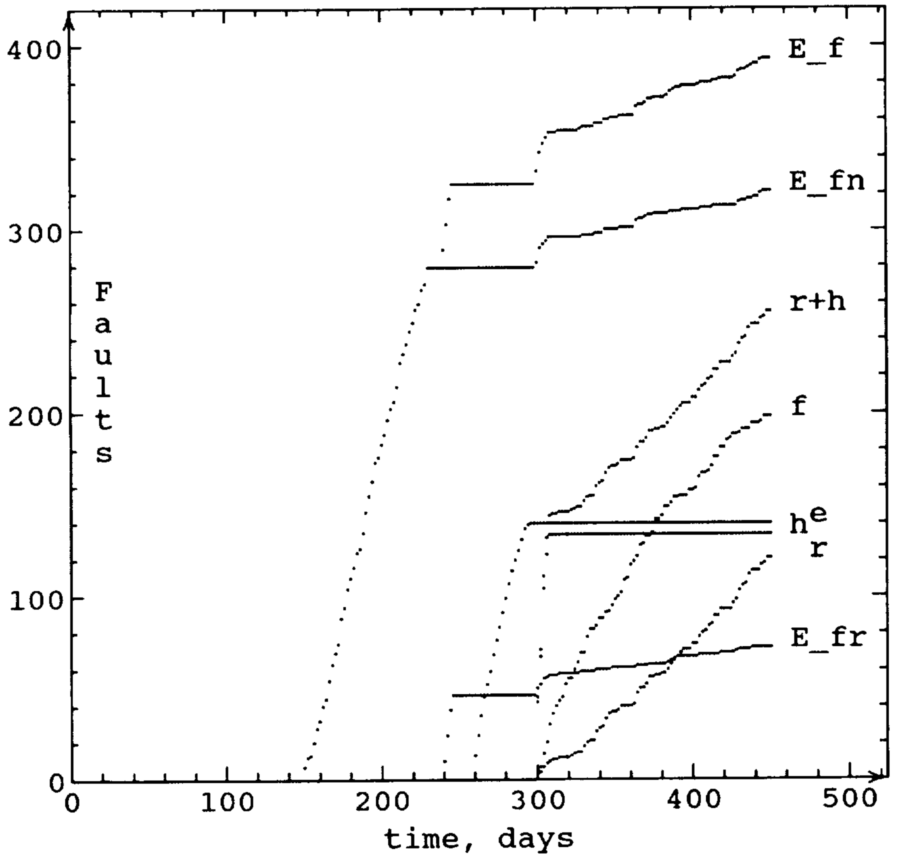

Figure 4: Fault Injection, Discovery, Correction, and Repair.

PAGE ________ INTENTIONALLY BLANK

6 CONCLUSION

Whether a `model` can be found that will simulate an actual project with the current $\mathcal{S}$oft$^{\rm r}$el rate functions is yet to be determined. However, even simulations using hypothetical data raise some serious questions in the mind of the author, not about what an actual project's `model` parameters will be, or what tuning should be applied to the rate functions, but on the fundamental issue of whether the "usual assumptions" listed earlier really do apply to the software reliability process. Are all the theoretical Poisson and exponential models fundamentally wrong? Real projects seem far more random than the simulated hypothetical fault process appears to be.[2]

One means for making simulations noisier, if that turns out to be warranted, is to randomly alter the schedule assignments at a fine level to depict irregular attention on the part of project personnel to reliability process matters. To the author's knowledge, such measures are not present in any of the "usual" mathematical models.

If regular schedules are assumed and if simulations are not as noisy as reality, then the underlying failure model cannot be Poisson, nor piecewise Poisson, nor perhaps even Markoff. Independence among faults and time increments implies that relative fluctuations behave as $O(1/\sqrt{n})$, a quantity that gets increasingly smaller for larger n. It seems likely, therefore, that simulation of document and code construction and integration processes will require irregular schedule models.

If irregular schedules are assumed, then the processes can be made to be as noisy as desired merely by matching process irregularities with schedule assignments. By linking nonhomogeneous piecewise-Poisson processes with randomly scheduled resources, we may be able to simulate actual highly fluctuating projects. That possibility will be explored and incorporated when, and if, later studies indicate that it is necessary.

At this point in time, of course, these questions are mere concerns and the commentary, mere speculation. The construction phase of $\mathcal{S}$oft$^{\rm r}$el has just completed its first prototype form. Validation of the simulation technique, the $\mathcal{S}$oft$^{\rm r}$el tool, and `model` parameters are yet to come. At this time, however, enough theory seems to be at stake to warrant review, as a first part of that validation.

Metric data collection of document size, code size, defect, failure, and fault data from open literature, and from industry and Jet Propulsion Laboratory (JPL) sources, has begun, but no significant inference of project parameters in the `model`, nor comparisons of measured project behavior against the simulator results, have been made at this time.

Future collaborations with industry, other National Aeronautics and Space Administration centers, and with JPL colleagues are expected. Through such

[2]This is an impression only. No effort has been made yet to validate this conjecture.

PRECEDING PAGE BLANK NOT FILMED

collaborations will come refinements to the simulation technique, better understanding and predictability of the reliability process, improvements to the *S*oft$^{\mathrm{r}}$el tool, and better characterization of what real project **models** are like.

In conclusion, this publication argues that we can simulate whatever we perceive reliability processes to be, including processes with multiple event categories, complex subprocess interdependencies, highly dimensional factors of influence, and schedule and resource dependencies. Evidence demonstrating that this can be done has been presented, subject to a few fundamental assumptions, such as the independence of events (e.g., failures) in non-overlapping time intervals. The assumptions made here are certainly weaker than those underlying the "usual" prediction models.

The initial *S*oft$^{\mathrm{r}}$el prototype has further assumed, for simplicity only, that defect and fault injections and removals take place with constant hazards per defect and fault. Other hazard functions will be investigated once sufficient project data have become available. The simulator has been designed so that its rate functions can be rather easily changed. Future reports will be made on such refinements when they occur.

7 REFERENCES

[1] Mellor, P, "Software reliability modeling: the state of the art," **Information and Software Technology**, Vol. 29, No. 2, March, 1987.

[2] Jelinski, Z., and Moranda, P. B., "Software reliability research," in **Statistical Computer Performance Analysis**, Freiburger, W., *Ed.*, Academic Press, New York, NY, 1972, pp. 465-484.

[3] Dale, C. J., "Software reliability evaluation methods," British Aerospace Dynamics Group, Rep. ST-26750, 1982.

[4] Musa, J., et al., **Software Reliability**, McGraw-Hill Book Co., New York, NY, 1987.

[5] Shooman, M. L., **Software Engineering**, McGraw-Hill Book Co., New York, NY, 1983, pp. 296-403.

[6] Abdel-Ghaly, A. A., et al., "Evaluation of Competing Software Reliability Predictions," **IEEE Trans. on Software Engineering**, Vol. SE-12, No. 9, September 1986, pp. 950-967.

[7] Levendel, Y., "Defects and Reliability Analysis of Large Software Systems: Field Experience," **Proc. Nineteenth International Fault-Tolerant Computing Symposium**, Chicago, IL, June, 1989.

[8] Papoulis, A., **Probability, Random Variables, and Stochastic Processes**, McGraw-Hill Book Company, New York, NY, pp. 534-551.

[9] Kreutzer, W., **System Simulation: Programming Styles and Languages**, International Computer Science Series, Addison-Wesley Publishing Co., Menlo Park, CA, 1986.

[10] Roberts, N., et al., **Introduction to Computer Simulation**, Addison-Wesley Book Co., Reading, MA, 1983.

[11] Knuth, D., **The Art of Computer Programming: Semi-Numerical Algorithms**, Addison-Wesley Book Co., Reading, MA, 1970, pp. 550-554.

[illegible]

APPENDIX

HYPOTHETICAL EXAMPLE

```
{ (05/31/90)                                                (hypothet.prj)
*************************************************************************
*      Copyright 1990  (C) California Institute of Technology           *
*      All rights reserved. U. S. Government sponsorship under NASA     *
*      Contract NAS-918 is acknowledged.                                *
*                                                                       *
*                      Robert C. Tausworthe                             *
*                      Jet Propulsion Laboratory                        *
*                      Pasadena, CA 91109                               *
*                                                                       *
*************************************************************************

                SOFTWARE PROJECT RELIABILITY PARAMETERS

NOTE: The appearance of digits, plus signs, minus signs, and periods in
this file is limited to occurrences in numbers only, or else within text
surrounded by braces, as in this note. Text and other characters
outside braces are ignored.  The numbers appearing in this file are
recognized in sequential order.                                         }

model sampling time interval, days:                                  2.
work fraction, workdays/schedule day/staff:                          1.

new documentation units:                                           500.
reused documentation units:                                        250.
reused documentation units deleted:                                 50.
reused documentation units added:                                   25.
reused documentation units altered:                                 20.

new documentation build rate (doc units/workday):                    2.5
reused documentation acquisition rate (doc units/workday):          10.
reused documentation deletion rate (doc units/workday):             10.
reused documentation addition rate (doc units/workday):              2.5
reused documentation alteration rate (doc units/workday):            2.5

new documentation defect generation rate (defects/doc unit):         0.3
reused document initial defect content (defects/doc unit):           0.03
reused document deletion defect rate (defects/doc unit):             0.3
```

PRECEDING PAGE BLANK NOT FILMED

```
reused document addition defect rate (defects/doc unit):             0.3
reused document alteration defect rate (defects/doc unit):           0.3
document hazard units added/removed per defect (hazard/defect):      1.

new documentation inspection fraction (inspected/total units):       0.9
reused document inspection fraction (inspected/total units):         0.9
document inspection rate (doc units/workday):                       11.
document inspection adequacy (defects found/defects present):        0.6

defect correction rate (defects/workday):                            2.
defect correction adequacy (corrections/attempt):                    1.
defect correction reinsertion rate (new defects/attempt):            0.3
documentation deleted per correction (doc units/attempt):            1.
documentation added per correction (doc units/attempt):              1.
documentation altered per correction (doc units/attempt):            1.

new code units:                                                  10000.
reused code units:                                                5000.
reused code units deleted:                                        1000.
reused code units added:                                           500.
reused code units altered:                                         250.

new code build rate (code units/workday):                           50.
reused code acquisition rate (code units/workday):                5000.
reused code deletion rate (code units/workday):                    200.
reused code addition rate (code units/workday):                     50.
reused code alteration rate (code units/workday):                   50.

new code fault generation rate (faults/code unit):                   0.02
reused code initial fault content (faults/code unit):                0.001
reused code deletion fault rate (faults/code unit):                  0.02
reused code addition fault rate (faults/code unit):                  0.02
reused code alteration fault rate (faults/code unit):                0.02
fault rate due to defective documentation (faults/defect):           0.9
fault rate due to missing doc (faults/code unit/doc fraction):       0.05
code hazard units added or removed per fault (hazard/fault):         1.

new code inspection fraction (inspected/total units):                0.9
reused code inspection fraction (inspected/total units):             0.9
code inspection rate (code units/workday):                         143.
code inspection adequacy (faults found/faults present):              0.6

fault correction rate (faults/workday):                              9.
fault correction adequacy (corrections/attempt):                     0.9
fault correction reinsertion rate (faults/attempt):                  0.2
code deleted per correction (code units/attempt):                   10.
code added per correction (code units/attempt):                     10.
code altered per correction (code units/attempt):                   10.
```

```
test case generation rate (test units/workday):                    50.
test case utilization rate (test units/resource day):               8.

fault failure rate (failures/resource day/faults per code unit):  300.
miss code failure rate (failures/resource day/miss code frac):    300.

failure observation rate (observed/occurred):                       0.9
failure outage rate (outages/failure):                              0.1
outage rate (outage days/outage failure):                           0.5

failure analysis rate (failures analyzed/workday):                 10.
fault identification rate (faults recognized/failure analyzed):     0.9

fault repair rate (fault repairs/workday):                          2.
fault repair adequacy (true repairs/attempt):                       0.9
fault repair reinsertion rate (new faults/attempt):                 0.3

repair validation rate (attempted repairs/workday):                10.
validation adequacy (detections/validation/misrepair):              0.9

retest rate (retested faults/workday):                              8.
retest adequacy (detected misrepairs/retest/misrepair):             0.9

========================================================================

                        SCHEDULE

start, finish,              activity,   staff, resource units/day
   0,     90, DOC_CONSTRUCTION    0,     2.5,              0.0
  90,    110, DOC_INTEGRATION     1,     2.5,              0.0
 110,    130, DOC_INSPECTION      2      2.5,              0.0
 130,    150, DOC_CORRECTION      3,     2.5,              0.0
 150,    240, CODE_CONSTRUCTION   4,     2.5,              0.0
 240,    250, CODE_INTEGRATION    5,     2.5,              0.0
 260,    300, CODE_INSPECTION     6,     2.5,              0.0
 300,    310, CODE_CORRECTION     7,     2.5,              0.0
 300,    450, TEST_PREPARATION    8,     0.4,              0.0
 300,    450, TESTING             9,     0.4,              1.0
 300,    450, FAULT_ID           10,     0.4,              0.0
 300,    450, REPAIR             11,     0.4,              0.0
 300,    450, VALIDATION         12,     0.4,              0.0
 300,    450, RETESTING          13,     0.4,              1.0
```

[illegible]

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 91-7	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A General Software Reliability Process Simulation Technique		5. Report Date April 1, 1991
		6. Performing Organization Code
7. Author(s) Robert C. Tausworthe		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No.
		11. Contract or Grant No. NAS7-918
		13. Type of Report and Period Covered External Report JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code RE156 BK-506-59-11-01-00
15. Supplementary Notes		

16. Abstract

This report describes the structure and rationale of the generalized software reliability process, together with the design and implementation of a computer program that will simulate this process. Given assumed parameters of a particular project, the users of this program are able to generate simulated status timelines of work products, numbers of injected anomalies, and the progress of testing, fault isolation, repair, validation, and retest. Such timelines are useful, in comparison with actual timeline data, for validating the project input parameters, and for providing data for researchers in reliability prediction modeling.

17. Key Words (Selected by Author(s)) Mathematical and Computer Sciences Computer Programming and Software Operations Research Statistics and Probablitiy		18. Distribution Statement Unclassified; unlimited	
9. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 53	22. Price

PRECEDING PAGE BLANK NOT FILMED