A Generalized Software Reliability Process Simulation Technique and Tool

Robert C. Tausworthe †
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
<tausworthe@isd.jpl.nasa.gov>
(818)306-6284

Michael R. Lyu
Bellcore
445 South Street
Morristown, NJ 07962
<lyu@bellcore.com>
(201)829-3999

Abstract

This paper describes the structure and rationale of the generalized software reliability process and a set of simulation techniques that may be applied for the purpose of software reliability modeling. These techniques establish a convenient means for studying a realistic, end-to-end software life cycle that includes intricate subprocess interdependencies, multiple defect categories, many factors of influence, and schedule and resource dependencies, subject to only a few fundamental assumptions, such as the independence of causes of failures. The goals of this research are dual: first, to generate data for truly satisfying the simplified assumptions of various existing models for the purpose of studying their comparative merits, and second, to enable these models to extend their merits to a less idealized, more realistic reliability life cycle. This simulation technique has been applied to data from a spacecraft project at the Jet Propulsion Laboratory; results indicate that the simulation technique potentially may lead to more accurate tracking and more timely prediction of software reliability than obtainable from analytic modeling techniques.

1: Introduction

Software reliability has been the subject of wide study over the past 20 years. At least 40 different models have been published in the literature so far[1]. The primary focus of these studies has been on proposing, analyzing, and evaluating the performance of models that assess current reliability and forecast future operability from observable failure data using statistical inference techniques. However, none of these models extends over the entire reliability process; most tend to focus only on failure observance during testing or operations. Moreover, none of these reliability models has emerged as ''the best'' predictor in all cases[2].

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 true underlying reliability process(es). It is conceivable that the basic nature of the failure process(es) may differ among individual software developments.

This paper proposes a general simulation technique that removes many current reliability modeling assumptions, and expands the reliability process to encompass the *entire* software life cycle. The usual assumptions for reliability modeling are:

(1) Testing (or operations) randomly encounters failures.

(2) Failures in non-overlapping time intervals are independent.

(3) The test space ''covers'' the use space (i.e., the operational profile).

(4) All failures are observed when they occur.

(5) Faults are immediately removed upon failure, or not counted again.

(6) Execution time is the relevant independent variable.

In particular, the second assumption above can be weakened to

(2) Faults produce independent failures.

and the final four assumptions are not necessary to the technique presented here at all. The degree of commonality among test space and use space is rarely known, but can be modeled, if needed. Simulation can mimic the failure to observe an error when it has, in fact,

† Part of the work reported in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

occurred, and, additionally, mimic any system outage due to an observed failure. Furthermore, simulation can easily distinguish those faults that have been removed and those that have not, so multiple failures from the same unremoved fault can be readily reproduced. Finally, while execution time is pertinent to some activities in the software life cycle, it is not appropriate to all; simulation can translate all model-pertinent times to wall-clock (or calendar) time by appropriate use of work load, computer utilization, and other resource schedules. This composite process is embodied in a Monte Carlo simulation tool, *SoftRel*[3], available through NASA's Computer Software Management Information Center (COSMIC)[4].

The remaining paper is organized as follows: Section 2 provides the basis for simulating the software reliability process; Section 3 briefly describes the structures and interactions of the reliability simulation package *SoftRel*; Section 4 presents a case study in which the implemented simulation technology was applied to a real-world project. Conclusions and future directions are presented in Section 5.

2: Simulation Building Blocks

2.1: Discrete Event Simulation Framework

The fundamental assumption of reliability process simulation is that every stochastic event is the result of an underlying instantaneous conditional event-rate random process. The most popular example of a conditional event-rate random process is the classic failure process[5].

A conditional event-rate process is one for which the probability that an event occurs in the interval $(t, t + dt)$, given that it had not occurred prior to time t, is equal to $\beta(t)\, dt$ for some function $\beta(t)$. The statistical behavior of this process is well-known: The probability that an event ε will have occurred prior to a given time t is related by the expression

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

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

Of special interest here are discrete event-count processes that merely record the occurrences of rate-controlled events over time. The function $\beta_n(t)$ denotes the conditional occurrence rate, given that the nth event has already occurred by the time t. The integral of $\beta_n(t)$ is $\lambda_n(t)$. These processes are termed non-homogeneous when $\beta_n(t)$ depends explicitly on t.

One important event-rate process is the discrete Markoff process[5]. A Markoff process is said to be homogeneous when its rate function is sensitive only to time differences, rather than to absolute time values. The notation $\beta_n(t)$, in these cases, signifies that t is measured from the occurrence time t_n of the nth event.

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

$$\bar{n} = \lambda(0, t)\ ; \quad \sigma^2 = \lambda(0, t) \qquad \text{(Eq.2)}$$

$$\frac{\sigma}{\bar{n}} = 1 / \sqrt{\lambda(0, t)} = 1 / \sqrt{\bar{n}} \qquad \text{(Eq.3)}$$

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

One may note from (Eq. 3) that as $\bar{n}$ increases, the percentage deviation of the process decreases. In fact, any event process with independence among events in non-overlapping time intervals will exhibit relative fluctuations that behave as $O(1/\sqrt{\bar{n}})$, a quantity that gets increasingly smaller for larger $\bar{n}$. This trend signifies that Poisson and Markoff processes involving large numbers of event occurrences will tend to become percentage-wise relatively calm. If physical processes appear to be very irregular, then it will not be possible to simulate them using independent-increment assumptions.

There is a sense in which the NHPP form is inappropriate for describing the overall software reliability profile. Reliability of software grows only as anomalies are discovered and repaired, and these events occur only at a finite number of times during the life cycle. The true hazard rate presumably changes discontinuously at these times, whereas the NHPP rate changes continuously. However, recent work[6] suggests that it is not possible to distinguish between an event-count Markoff process with a discontinuous rate function and an overall NHPP reliability growth model with an appropriately defined continuous rate function, based merely on examination of a single realization of either process. In any case, the event-count Markoff model of software reliability is more general than the NHPP form, in that there is no assumption that its cumulative rate λ is independent of n or t_n.

2.2: Event Simulation

The very definition of conditional event-rate processes suggests the rather simple computer simulation illustrated in the following C language segment:

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

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

/* the event has occurred a number of times */
```

The dt in such simulations must always be chosen such that the variations in $\beta(t)$ over the incremental time intervals $(t, t + dt)$ are negligible, and such that $\beta(t)\, dt < 1$ (so that the instantaneous event probability does not reach unity)[7]. In the code segment above, `chance(T)` compares a [0,1)-uniform `random()` value with `T`, thus attaining the specified instantaneous probability function. The form of `beta(events, T)` acknowledges that the event rate function may change over time and may be sensitive to the number of event occurrences up to the current time.

The above illustration of simulation is simple, and yet very powerful. For example, some published analytic models treat (or approximate) the overall reliability growth as a NHPP in execution time, while others focus on Markoff execution-time interval statistics. Many of these differ only in the forms of their rate functions[1] [8]: Some examples are

1. The Jelinski-Moranda model[9] deals with adjacent time-interval subprocesses in which $\beta_n(t) = \phi\,(n_0 - n)$, where n_0 is the (unknown) number of initial software faults and ϕ is the per-fault failure rate.

2. The Goel-Okumoto model[10] deals with the overall reliability growth process, in which $\beta(t) = n_0\, \phi\, exp(-\phi t)$, where n_0 and ϕ are constant parameters. It has been shown[6] that this model produces results very much like the Jelinski-Moranda model with $n = n_0(1 - exp(-\phi t))$.

3. The Musa-Okumoto model[11] describes the overall reliability growth process, in which $\beta(t) = \beta_0/(1 + \theta t)$, where β_0 is the initial failure rate and θ is a rate decay factor. Both β_0 and θ are constant parameters.

4. The Duane model[12] is an overall reliability growth model with $\beta(t) = kbt^{b-1}$, where k and b are constant parameters.

5. The Littlewood-Verrall inverse linear model[13] is an overall reliability growth model with $\beta(t) \tilde{=} \phi / \sqrt{1 + kt}$, where ϕ and k are constant parameters.

6. The Yamada delayed S-shape model[14] is yet another overall reliability growth model, with $\beta(t) = \phi\gamma t\ exp(1 - \gamma t)$, where ϕ (the maximum failure rate) and γ are constant parameters.

Simulating the reliability process underlying these models is straightforward. Interested readers please refer to[15] for details.

2.3: Poisson Process Simulation

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

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

where `random_poisson(x)` is a subprogram that produces a Poisson-distributed random value when passed the parameter `x`. An algorithm for generating Poisson random numbers may be found in[16].

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

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

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

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

2.4: Multiple Event Processes

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

$$\beta_n(0, t) = \sum_{i=1}^{f} \beta_n^{[i]}(0, t). \qquad \text{(Eq.4)}$$

When event occurrences in non-overlapping intervals are independent, each (t_a, t_b) interval is governed by a non-homogeneous Markoff process with rate $\beta_n(t, t_n)$.

$$\beta_n(t, t_n) = \sum_{i=1}^{f} \beta_{n_i}^{[i]}(t, t_{n_i}) \qquad \text{(Eq.5)}$$

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

2.5: Multiple Categories of Events

If the set of events $\{\varepsilon_i : i = 1, \cdots, n\}$ that were classed together above are now partitioned into categorized subsets according to some given differentiation criteria (as for example, faults distinguished as being *critical*, *major*, or *minor*), then the partitioning of events into categories likewise partitions their rate functions into corresponding categories, and equivalently, the bracketed indices of the rate functions into sets of integers.

When an event occurs, the algorithm of Subsection 2.4 produces the index of a rate function. Finding this index among the categorized subsets of integers relates the event to the distinguished category of occurrences. The behavior of multiple categories of events is thus easily simulated by changing from a single event counter, `events`, to an array of event counters, `events[ ]`, and altering the program as follows:

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

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

2.6: Other Event Processes

In the software life cycle, it is often the case that, if an event of one type occurs, then there is a uniform probability $p < 1$ that another event of a different type will be triggered. (For example, suppose that for each unit of code is generated, there is a probability p that a fault is created.) If there are n events of the first type, then the k events of the second type are governed by the binomial distribution function, which is also easily simulated[16].

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

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

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

2.7: General Event-Rate Processes

The simulation method of this paper is more general than is required for mere production of Markoff processes and NHPPs. Since the algorithm of Subsection 2.2 springs directly from the definition, the method is capable of simulating all event-rate random processes.

It thus is possible to simulate life cycle activities that may have event count dependencies between non-overlapping time intervals and rate functions that depend on variable schedules and other irregularities over time. Whenever event functions produce homogeneous Markoff processes in a piecewise fashion, then the event processes simulated during each of these segments will follow the piecewise-Poisson approximation. The programs presented above are thus capable of simulating a much more general and realistic reliability process than has been hypothesized by any analytic model known to the authors.

3: Structure of the Simulation Tool

3.1: Overall Simulation Context

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

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

The current input to *SoftRel* consists of a single file that specifies the `dt` time slice, about 70 traits of the software project and its reliability process, and a list of activity, schedule, and resource allocations. Internally, these form a data structure called the `model`. Also internally, the set of status monitors at any given time are stored in a data structure called `facts`, which records the overall clock time, the time and resources consumed by each activity (42 measures in total), and a snapshot of 48 measures of project status. The output from *SoftRel* is a single file containing the series of `facts` produced at each `dt` interval of time.

SoftRel simulates two types of failure events, namely, defects in specification documents and faults in code. Figure 1 shows the execution context of *SoftRel*.

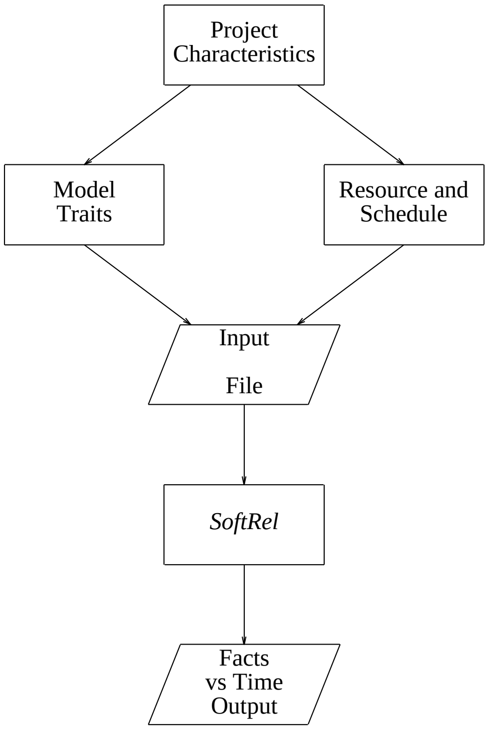

Figure 1: ***SoftRel*** **Execution Context**

3.2: The Major Components of the Simulator

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

In a typical software engineering life cycle, several interrelated software reliability subprocesses are taking place concurrently. The activities in these subprocesses are characterized by 14 major components in the simulator, with appropriate staffing and resource levels devoted to each activity:

(1) *Document Construction*: Document generation and integration are assumed to be piecewise-Poisson approximations with constant mean rates per work-

day specified in the `model`, not to exceed the goal values. Defects are injected at a constant probability per documentation unit. At each injection of a defect, the document hazard increases according to the defect detection characteristic.

(2) *Document Integration*: Document integration consists of 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 approximation 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.

(3) *Document Inspection*: Document inspection is a goal-limited piecewise-Poisson approximation of a type similar to document construction. 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. The defect discovery rate is assumed to be proportional to the current accumulated document hazard and the inspection efficiency.

(4) *Document Correction*: Defect corrections are produced at a rate determined by the staff level and the attempted-fix rate given in the `model`; actual corrections take place according to the defect-fix adequacy, not to exceed the actual number of defects discovered (a goal-limited binomial situation). Attempted fixes can also inject new defects and can change the overall amount of documentation via the numbers of documentation units deleted, added, and changed.

(5) *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 fault density that may occur as a normal consequence of coding, but also by the density of undiscovered defects in documentation, and by the amount of missing documentation. Each fault injected increases the code hazard. But whereas document defects are only found by inspection, code faults may be found by both inspection and testing, and at different rates.

(6) *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. Each fault increases the code hazard.

(7) *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 as-yet undiscovered faults.

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

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

(10) *Testing*: The testing activity simulation has two parts: If a test outage is in effect, the outage time indicator decrements and the time and effort increment. If an outage is not in effect, failures occur at the `model`ed rate; the number observed is computed as a binomial process that is regulated by the probability of observation. 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.

(11) *Fault Identification*: The total number of failures analyzed may not exceed the number of failures observed. Failures are analyzed at a mean constant rate per workday. The identification of faults is limited in number to those still remaining in the system. The isolation process is regulated by the fraction of faults remaining undiscovered, the adequacy of the analysis process, and the probability

of faithful isolation.

(12) *Fault Repair*: The number of attempted repairs may not exceed the number of faults identified by inspections and testing, less those corrected after inspection, plus those identified for rework by validation and retesting. Of those attempted, a `select` number will really be repaired, while the rest will mistakenly be reported as repaired. Repairs are assumed here to be made on faults identified for rework first. A `select` number of new faults may be created by the attempt, and code units may be altered (deleted, added, or changed).

(13) *Validation of Repairs*: The validation of attempted repairs takes place at an assumed mean constant rate per workday. The number of faulty repairs detected is a `select` number determined by the probability that validation will recognize an unrepaired fault when one exists and the probability that unrepaired faults are among those attempted repairs being validated (the repair adequacy).

(14) *Retesting*: Retesting takes place at a mean constant number of retests per workday and consumes computer resources at the scheduled rate per day. No new test cases are generated (or consumed), since the original test cases are assumed available for regression.

3.3: Simulator Input and Output

It is beyond the scope of this paper to describe each of the 70 input `model` parameters and the 90 output `facts` parameters. Interested readers will find these described more fully in[3]. The input file additionally contains a list of staffing and computer resource packets, each of which allocates resources to specified activities and time slots. 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. At least 14 schedule packets are needed to allocate resources and time slots to each of the 14 assumed reliability process activities. More packets may appear when an activity repeats or has a non-constant resource allocation profile.

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

The reliability process embodied in the prototype is meant to be fairly comprehensive with respect to what really transpires during a software development. The simulator therefore requires parameters relating to the ways in which people and processes interact. The large number of parameters in the simulator might, at first, seem to present an overwhelming, impractical barrier to modeling, but it must be remembered that the true reliability process is even more complex. It was felt that the number of parameters used was the least that would be capable of reproducing the realism hoped for. Reducing the number of parameters might either reduce the fidelity of the simulation or the generality of the reliability process model. This belief may change after sufficient experimentation has taken place, whereupon selective alteration of the parameters and rates may be indicated.

If projects do not have sufficient data about past projects to give values to certain parameters, then sensitivity analyses using *SoftRel* can indicate which are the most influential and thereby where a metrics effort may prove most useful in reliability management. Alternatively, users may simplify the model to focus only on one or two activities at a time by making some of the parameters inactive. This may be done by assigning typical or default values (usually 0 or 1) to them, thereby reducing the number of measured parameters to only those that are deemed pertinent and realistic within the project context.

4: Applications: A Project Case Study

This Section describes the application of *SoftRel* to a real-world project. Project data and parameters from a subsystem of the Galileo Flight Project at the Jet Propulsion Laboratory were collected as a case study for evaluating the simulation technique. The remainder of this Section describes the project, applies the simulation technique, and compares the results with those obtained from several traditional reliability models.

4.1: Project Description

Galileo is an outer planet spacecraft project that began at the start of fiscal year 1977, a mission that was originally entitled ‘‘Jupiter Orbiter and Probe,’’ or JOP. Unlike previous outer solar system missions, the Galileo orbiter was intended to remain in Jovian orbit for an extended interval of time. This would allow observations of variations in planetary and satellite features over time to augment the information obtained by single-observation opportunities afforded by previous fly-by missions. Galileo was launched in October of 1989, and will reach the Jovian system in 1995.

There are two major on-board flight computers in the Galileo spacecraft: The Attitude and Articulation Control Subsystem (AACS), and the Command and Data System (CDS). A significant portion of each of these systems is embodied in software. This case study focuses on the CDS software reliability profile.

The CDS flight software is characterized as real-time embedded software, written in 17,000 lines of assembly language code (16.5K lines new, 500 lines reused), with about 1400 pages of documentation (1300 pages new, 100 pages reused), produced over a period of approximately 1500 days (300 weeks). The project recorded and tracked failures during the software testing period.

4.2: Parameter Estimations and Results

This Subsection presents a simulation of an end-to-end development project based on data from the Galileo CDS project. Some of the CDS project parameters were taken from project records; other values were estimated by personnel within the project; the remaining values were chosen by the authors as being probably typical of this project's behavior, but for which there were no immediately available data. Believed-typical values were adopted, for example, for parameters related to the injection of faults in the correction and repair processes. None of the `model` input parameters was set to zero.

Thus, even though few verifiable `model` parameters were available outside the software testing phase, an entire plausible hypothetical model was nevertheless formed in order to illustrate simulation of an end-to-end reliability process. For lack of better development life cycle data, all CDS activities other than testing (e.g., construction, inspection, and anomaly removal) were presumed to take place serially, merely to observe what their simulated behaviors would be. This overall study also presumed that each activity took place without resource and schedule variations, in order to view typical Markoff reliability behavior. The `model` parameters that were used are available from the authors upon request.

Figures 2 through 5 show the simulated documentation, code, defect, and fault profiles of the software, sampled every 10 days. Of particular note are behaviors of the documentation, code, injected defects, and injected faults (precisely those activities where no project data exists). Because the numbers of units are comparatively large, the relative irregularity levels are low, as predicted from (Eq. 3). Although there is no actual CDS data contesting this behavior, it seems doubtful that this almost-linear profile reflects actuality.

If this doubt were confirmed, then an assumption of a homogeneous reliability process would clearly be proved invalid. A more realistic extension to the case study would be to introduce irregular schedules, since it is known that people rarely dedicate their time exclusively to one single activity at a time. If actual CDS schedule information were available and entered into the `model`, these processes could easily have appeared to be more irregular.

Figure 2 shows that the volume of documentation units (`DU_t`) did reach its goal, but in this case, only about 63% of the documentation was actually inspected (`DI_t`), even though the `model` placed a goal of 95% on inspection. This is an instance where too little resource was allocated to the inspection process. More resources would have been required to reach the goal. The effects of correcting defects on page count are not visible. The second rise in documentation is due to the integration of the reused 100 pages.

Figure 3 similarly shows that the volume of code units (`CU_t`) 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, detection, and removal of defects (`E_d, D, d`), shown in Figure 4, are a little noisier than documentation and code production, but not much. All the detected defects were corrected (`D = d`), but a sizable number of defects were inserted during the correction period (days 520-580). Finally, more than 100 defects were left in the documents.

The fault activity is shown in Figure 5. It exhibits the noisiest behavior of all, but is still fairly regular. The initial rise in injected faults (`E_f`) is due to construction; the second rise, due to integration, is not visible; 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 1500-day project, about 7 faults per kiloline of code (`e`) had been found in inspections and corrected (`h`), and about 22 faults per kiloline of code (`f`) had been uncovered by testing and removed (`r`); the fault density at delivery was about 0.2 faults per kiloline of code.

Although the final fault discovery count is considered to be accurate, the time profile of the simulation results do not appear to be as irregular as the actual project data. On the basis of this case study, it appears that the simulation of all reliability subprocesses will require the use of non-homogeneous event-rate models that reflect irregular workloads and schedules of life cycle activities. An example of this is given in the next Subsection.

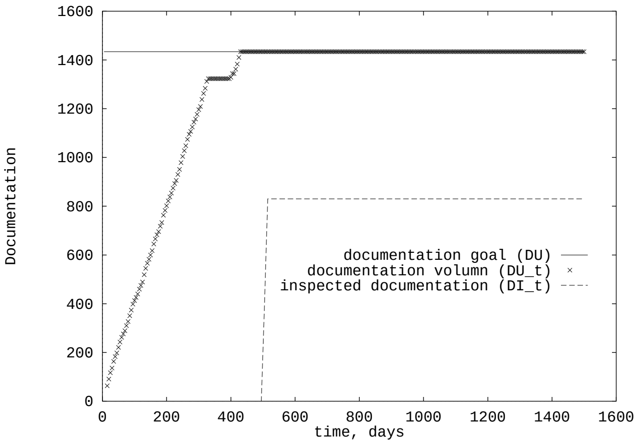

Figure 2: CDS Simulated Documentaion Development

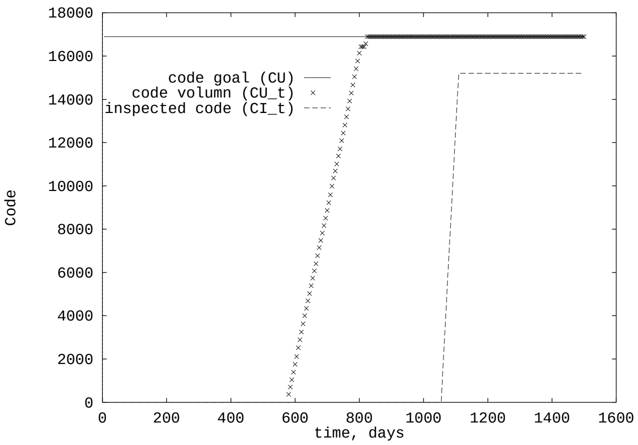

Figure 3: CDS Simulated Code Development

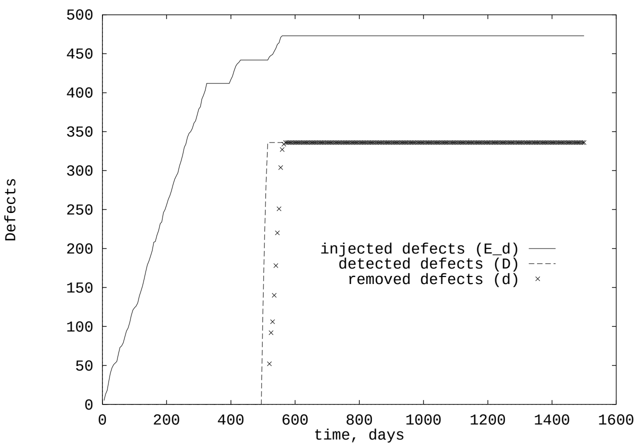

Figure 4: CDS Simulated Defect Discovery and Correction

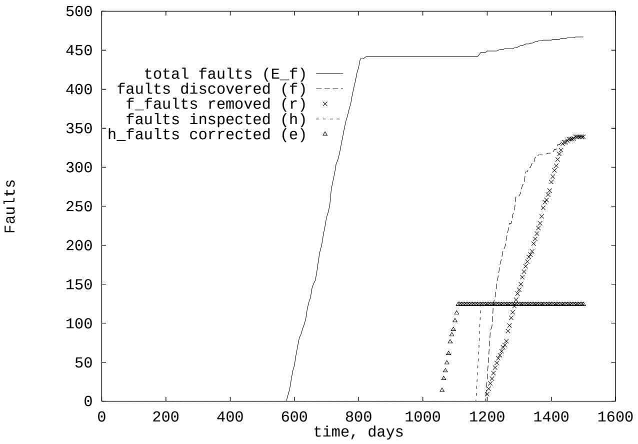

Figure 5: CDS Simulated Fault Accumulation

4.3: Comparisons with Other Models

To simulate the details of Galileo CDS testing activity, its testing phase was separated into five subactivities with constant staffing, but having irregular CPU and schedule allocations, as shown in Table 1. These schedule parameters were obtained by ‘‘eye-ball regression’’ of the simulator output against project data. The fit appears adequate to describe the underlying nature of the failure process (an exact fit is not expected, since the failure process is considered random).

activity	total failures	begin week	end week	staff	CPU
functional test	90	0	5	2.0	0.4
feature test	60	5	13	2.0	0.4
operation test1	150	13	23	2.0	1.2
operation test2	25	23	33	2.0	1.0
operation test3	16	33	40	2.0	2.0

Table 1: Schedule of the CDS Testing

Figure 6 shows the field data and the results obtained from the piecewise-homogeneous simulation process, as well as those from three other models, Jelinski-Moranda Model(JM), Musa-Okumoto Model(MO), and Littlewood-Verrall Model(LV). For better visibility of process granularity, data is shown in the form of failures per week, rather than cumulatively. The JM, MO, and LV statistics were calculated to be "one-week-ahead" predictions, in which all the failure data up to a given week were used to predict the number of failures for the next week.

It can be seen from the figure that the simulation technique produced a very good early forecast that could have been used for tracking the reliability status during the entire testing phase. The forecast could have been calculated prior to the start of testing from schedule and resource plans. The other models above were inadequate to predict even one week ahead.

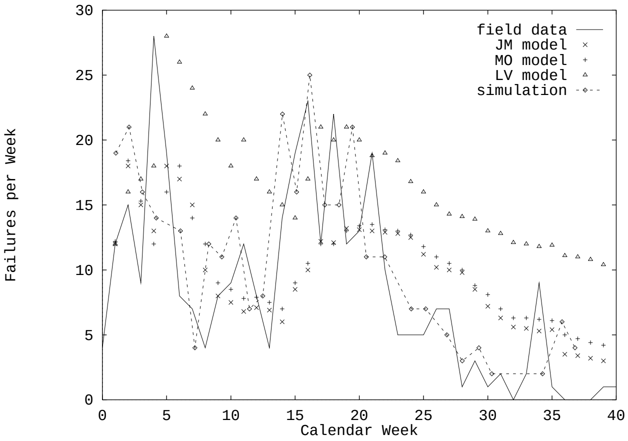

Figure 6: Various Predictions for CDS Failures per Week Data

5: Conclusions

Reliability modelers seldom seem to have the luxury of several realizations of the same failure process to test their hypotheses concerning the nature of a system's reliability. Nor are they ever provided with data that faithfully match the assumed natures of their models. Nor are they able to probe into the underlying error and failure mechanisms in a controlled way. Rather, they are faced with the problem of not only guessing the forms and particulars of the underlying error and failure random processes from the scant, uncertain data they possess, but also with the problem of best forecasting future failures from this single data set.

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

Simulation of a real-world project has proved the validity of the approach. It was shown that neither NHPP nor homogeneous Markoff event-count models adequately reproduced the statistical failure profile of an actual project. A non-homogeneous event-rate simulation model was demonstrated that produced very good early forecasts of reliability growth that would have proved adequate for process status assessment.

References

1. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability – Measurement, Prediction, Application*, McGraw-Hill Book Company, New York, New York, 1987.
2. M.R. Lyu and A. Nikora, ''Using Software Reliability Models More Effectively,'' *IEEE Software*, pp. 43-52, July 1992.
3. R. Tausworthe, ''A General Software Reliability Process Simulation Technique,'' Technical Report 91-7, Jet Propulsion Laboratory, Pasadena, California, March 1991.
4. R. Tausworthe, ''*SoftRel* Program NPO-1825 (CP-7814),'' *COSMIC catalog*, NASA, 1991.
5. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 1965.
6. D.R. Miller, ''Exponential Order Statistic Models of Software Reliability Growth,'' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 12-24, January 1986.
7. N. Roberts *et al.*, *Introduction to Computer Simulation*, Addison-Wesley, Reading, Massachusetts, 1983.
8. Y. Malaiya and N. Karunanithi, ''Predictatility Measures for Software Reliability Models,'' in *Proceedings COMPSAC-90*, pp. 7-12, Chicago, Illinois, October 1990.
9. Z. Jelinski and P.B. Moranda, ''Software Reliability Research,'' in *Statistical Computer Performance Evaluation*, ed. W. Freiberber, pp. 465-484, Academic, New York, 1972.
10. A.L. Goel and K. Okumoto, ''Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures,'' *IEEE Transactions on Reliability*, vol. R-28, pp. 206-211, 1979.
11. J.D. Musa and K. Okumoto, ''A Logarithmic Poisson Execution Time Model for Software Reliability Measurement,'' in *Proceedings Seventh International Conference on Software Engineering*, pp. 230-238, Orlando, Florida, 1984.
12. J.T. Duane, ''Learning Curve Approach to Reliability Monitoring,'' *IEEE Transactions on Aerospace*, vol. AS-2, pp. 563-566, 1964.
13. B. Littlewood and J.L. Verrall, ''A Bayesian Reliability Growth Model for Computer Software,'' *Journal Royal Statistics Society C*, vol. 22, pp. 332-346, 1973.
14. S. Yamada, M. Ohba, and S. Osaki, ''S-Shaped Reliability Growth Modeling for Software Error Detection,'' *IEEE Transactions on Reliability*, vol. R-32, pp. 475-478, December 1983.
15. R. Tausworthe and M.R. Lyu, ''Software Reliability Process Simulation,'' in *McGraw-Hill Software Reliability Engineering Handbook*, ed. M.R. Lyu, McGraw-Hill, New York, February 1995.
16. D.E. Knuth, *The Art of Computer Programming: Semi-Numerical Algorithms*, Addison-Wesley, 1970.