Submitted to the Performance Evaluation Journal, special issue on Tools for Performance Evaluation to be published in 1999

SREPT: Software Reliability Estimation and Prediction Tool *

Srinivasan Ramani[1], Swapna S. Gokhale[2], Kishor S. Trivedi[1]
[1]Center for Advanced Computing and Communication
Department of Electrical and Computer Engineering
Duke University, Durham, NC 27708-0291, USA
{sramani,kst}@ee.duke.edu
Phone : (919) 660-5269
Fax : (919) 660-5293

[2]Bourns College of Engg.
University of California
Riverside, CA 92521, USA
swapna@cs.ucr.edu

Abstract

Several tools have been developed for the estimation of software reliability. However, they are highly specialized in the approaches they implement and the particular phase of the software life-cycle in which they are applicable. There is an increasing need for a tool that can be used to track the quality of a software product during the software life-cycle, right from the architectural phase all the way up to the operational phase of the software. Also the conventional techniques for software reliability evaluation, which treat the software as a monolithic entity, are inadequate to assess the reliability of heterogeneous systems, which consist of a large number of globally distributed components. Architecture-based approaches are essential to predict the reliability and performance of such systems. This paper presents the high-level design of a *Software Reliability Estimation and Prediction Tool* (SREPT), that offers a unified framework consisting of techniques (including the architecture-based approach) to assist in the evaluation of software reliability during all phases of the software life-cycle.

*Supported in part by a contract from Charles Stark Draper Laboratory and in part by Bellcore as a core project in the Center for Advanced Computing and Communication

1 Introduction

Software is an integral part of many critical and non-critical applications, and virtually any industry is dependent on computers for their basic functioning. As computer software permeates our modern society, and will continue to do so at an increasing pace in the future, the assurance of its quality becomes an issue of critical concern. Techniques to measure and ensure reliability of hardware have seen rapid advances, leaving software as the bottleneck in achieving overall system reliability.

Software reliability is defined as *the probability of failure-free software operation for a specified period of time in a specified environment* [1]. Its evaluation includes two types of activities [10] : reliability *estimation* and reliability *prediction*.

- **Estimation** - This determines the current software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This is a measure of "achieved" reliability, and determines whether a reliability growth model is a good fit in retrospect.

- **Prediction** - This determines future software reliability based upon a fitted reliability growth model or other available software metrics and measures.

Various approaches have been proposed to achieve reliable software systems and these may be classified as [29] :

- **Fault prevention** - To avoid, *by construction*, fault occurrences.

- **Fault removal** - To detect, *by verification and validation*, the existence of faults and eliminate them.

- **Fault tolerance** - To provide, *by redundancy*, service complying with the specification in spite of faults having occurred or occurring.

- **Fault/failure forecasting** - To estimate, *by evaluation*, the presence of faults and the occurrence and consequences of failures.

Of these, fault/failure forecasting techniques have received widespread attention with the development of *software reliability growth models*. These models capture the process of fault detection and removal during the testing phase in order to forecast failure occurrences and hence software reliability during the operational phase. Many of these models have been encapsulated into tools [10] like

SMERFS, AT&T SRE Toolkit, SoRel, CASRE [16], RAT [38], M-*élopée* [39]. The problem with applying such tools effectively towards improving the quality of software is that they need data about the failures detected during the testing phase in order to estimate software reliability model parameters. Thus they assess reliability very late in the life-cycle of the software, and it may be too costly to take remedial actions if the software does not meet the desired reliability objective. Also, these tools do not take into account the architecture of the software. Though some techniques have been proposed [43] [44] [45] to perform architecture-based software reliability prediction, there is no special purpose tool available yet. There is thus a need for a tool that can track the quality of a software product and provide insights throughout the life-cycle of the software. In this paper we present the high level design of a tool which provides a unified framework encompassing several techniques to monitor the reliability of a software product during the entire software development process.

The rest of the paper is organized as follows. In the next section we provide the motivation for a new tool. Section 3 presents the high-level design of SREPT. Section 4 provides an illustration of the use of SREPT in estimating software reliability. In Section 5 we comment on SREPT's expected impact in the area of software reliability estimation and prediction. Section 6 concludes the paper by highlighting the features of SREPT that are presented in this paper.

2 Motivation

For state of the art research efforts to become best current practices in the industry, they ought to be made available in a systematic, user-friendly form. This factor has motivated the development of several tools for software reliability estimation and prediction. These tools can be broadly categorized into the following groups :

- Tools which use software metrics (product/process metrics) at the end of the development phase as inputs and either classify the modules into fault-prone or non-fault-prone categories, or predict the number of faults in a software module. Examples of such tools include the *Emerald* system [9], ROBUST [34] [35] and McCabe & Associates' Visual Quality Toolset (VQT) [42].

- Tools which accept failure data during the functional testing of the software product to calibrate a software reliability growth model based on the data, and use the calibrated model to make predictions about the future. SMERFS, AT&T SRE Toolkit, SoRel, CASRE [10],

RAT [38], M-élopée [39] and ROBUST [34] [35] are examples of such tools.

- Tools (ESTM [24] [25], for example) that utilize failure data and specified criteria to suggest release times for software.

However, these tools are highly specialized in the approaches they implement and the phase of the software life-cycle during which they are applicable. We present below the limitations of existing tools, and highlight the need for a new tool which offers a unified framework for software reliability estimation and prediction.

2.1 Limitations of Existing Tools

The *Emerald* system [9] uses the Datrix software analyzer to measure static attributes of the source code to collect about 38 basic software complexity metrics. Based on the experience in assessing previous software products, thresholds are set on these metrics to identify modules with *out-of-range* metrics as *patch-prone*, so that effort can be expended in improving the quality of such modules. The *Emerald* system can thus be used to determine the quality of a software product after the development phase, or pre-test phase. However it does not offer the capability of obtaining predictions based on the failure data collected during the testing phase.

On the other hand, tools [10] like SMERFS [19], AT&T SRE Toolkit, SoRel [20], CASRE [16] [17] [18], RAT [38], M-élopée [39] can be used to estimate software reliability using the failure data to drive one or more of the software reliability growth models (SRGM). These tools differ in the number of SRGMs they implement and the interfaces they provide the user. However they can only be used very late in the software life-cycle, and early prediction of software quality based on static attributes can have a lot of value. ROBUST [34] [35] is another tool that uses SRGMs to obtain software reliability predictions based on failure data. But, in addition, the tool can also provide an early estimate of the testing effort required, based on static metrics like the defect density, test phase the software is in, the maturity of the test team, number of lines of code, and so on. Thus it seeks to address one of the drawbacks of the other SRGM-based tools described above. Still, the tool assumes instantaneous repairs (an assumption inherent in most prevalent SRGMs and which may not be a good assumption for some failure data sets), and also lacks the ability to perform software architecture-based predictions.

Techniques to obtain the optimal software release times guided by the reliability estimates obtained from the failure data have also been encapsulated in tools. For example, the ESTM [24] [25] tool determines release times using a stochastic model for the detection of faults, and an economic

model to measure the trade-off between shipping and further testing. The stochastic model used in ESTM assumes that the times to find faults are i.i.d. exponentially distributed with rate, say μ. The economic model takes into account the cost of fixing a fault while testing, the cost of fixing a fault when found in the field, and the cost of testing up to some time t plus the cost of not releasing the software up to time t. The stopping rule for testing specifies the point at which the incremental cost of another day's testing exactly balances the expected net decrease in future expense due to faults that will be found in that day's testing. It also gives a probabilistic estimate on the number of remaining faults at that point. Another interesting feature is that the model used by ESTM can take into account new or changed noncommentary source lines while the testing effort is still in progress. But once again, ESTM addresses only a particular problem (though very important) in the software life-cycle.

Though some of the conventional techniques to assess software reliability are available in the form of tools, these techniques have been shown to be inadequate to predict the reliability of modern heterogeneous software systems. System architecture-based reliability prediction techniques have thus gained prominence in the last few years [11] [26]. Presently there exists no special purpose tool for architecture-based analysis.

For the sake of completion, it might be mentioned here that another class of tools that are useful aids during the software reliability assessment process are the *test coverage "measurement" tools* such as ATAC [40] and DeepCover [41]. ATAC is a test coverage analysis tool for software written in C, developed at Bellcore. The use of ATAC focuses on three main objectives: instrumenting the software (at compile time) to be tested, executing the software tests, and determining how well the software has been tested. DeepCover is a test coverage analysis tool for C/C++/Java code, developed by Reliable Software Technologies. Trace information that is generated by such coverage analysis tools can be used to obtain the architecture of the software [32]. Knowledge of the architecture of a software is essential to perform architecture-based software reliability predictions, as will be explained later in Section 3.2.

2.2 Need for a Unified Framework for Software Reliability Estimation

The above discussion highlights the need for a tool that offers a unified framework consisting of techniques to assist in the assessment of the reliability of the software during all phases of the software life-cycle. Features such as the ability to suggest optimal times to stop testing are highly desirable. The high-level design of such a tool, which we call *Software Reliability Estimation and*

Prediction Tool (SREPT), is presented in this paper.

SREPT combines the capabilities of the existing tools in a unified framework. In addition, it offers the following features :

- Provides a means of incorporating test coverage into finite failure NHPP (Non-Homogeneous Poisson Process) reliability growth models, thus improving the quality of estimation.
- It offers a prediction system to take into account finite fault removal times as opposed to the conventional software reliability growth models which assume instantaneous and perfect fault removal. The user can specify the fault removal rate which will reflect the scheduling and resource allocation decisions.
- It incorporates techniques for architecture-based reliability and performance prediction.

3 Design and Architecture of SREPT

This section presents the high-level design of SREPT. The block diagram in Figure 1 depicts the architecture of SREPT. As can be seen from the figure, SREPT supports the following two approaches to software reliability prediction - the *black-box*-based and the *architecture*-based approaches.

3.1 Black-Box Based Approaches

Black-box based approaches treat the software as a whole without considering its internal structure. The following measures can be obtained to aid in black-box predictions -

- *Software product/process metrics* - these include the number of lines of code, number of decisions, loops, mean length of variable names and other static attributes of the code, or characteristics of the process (example, skill level of the development team, criteria to stop testing)
- *Test coverage* - this is defined to be the ratio of potential fault-sites exercised (or executed) by test cases divided by the total number of potential fault-sites[1] under consideration [8].
- *Interfailure times data* - this refers to the observed times between failures when the software is being tested.

[1]A potential fault-site is defined as any structurally or functionally described program element whose integrity may require verification and validation via an appropriately designed test. Thus a potential fault-site could be a statement, a branch, etc.

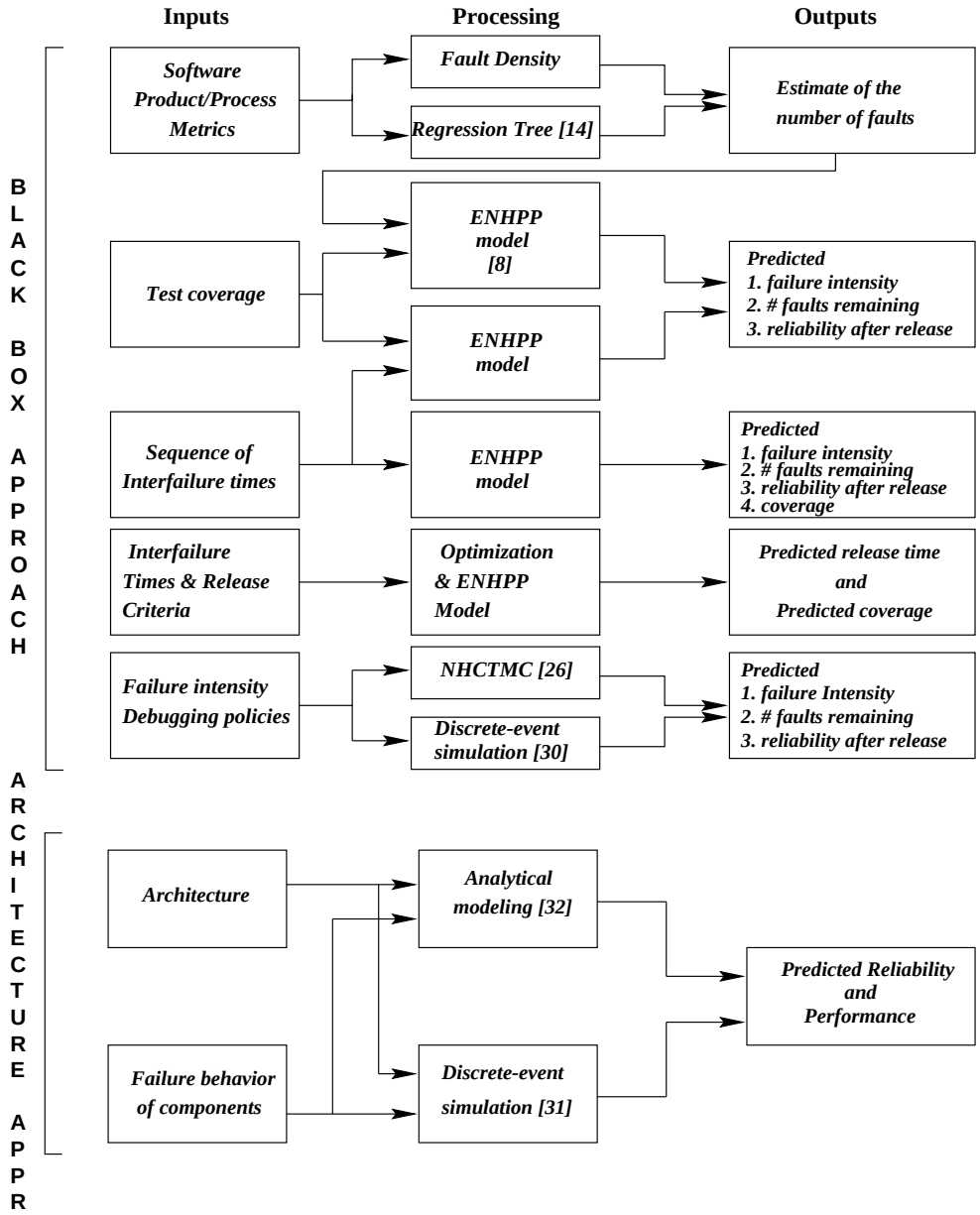

Figure 1: Architecture of SREPT

When **product/process metrics** are available, the total number of faults in the software can be estimated using the *fault density* approach [13] or the *regression tree* model [14]. This is because the product/process metrics are a quantitative description of the static attributes of a program and are closely related to the distribution of faults [46]. In the fault density approach, experience from similar projects in the past is used to estimate the fault density (FD) of the software as,

$$FD = \frac{total\ number\ of\ faults}{number\ of\ lines\ of\ code}$$

Now, if the number of lines of code in the current software is N_L, the expected number of faults can be estimated as,

$$F = N_L * FD$$

The regression tree model is a goal-oriented statistical technique [47], which attempts to predict the number of faults in a software module based on the static product/process metrics. Historical data sets from similar past software development projects is used to construct the tree which is then used as a predicting device for the current project. Software product/process metrics can thus be used to perform early prediction of the number of faults in the modules of a software product. This can help channel testing and validation efforts in the right direction [14].

Interfailure times data obtained from the testing phase can be used to parameterize the ENHPP (Enhanced Non-Homogeneous Poisson Process) model [8] to obtain the following estimates for the software -

- failure intensity,
- number of faults remaining,
- conditional reliability, and
- test coverage.

The ENHPP model provides a unifying framework for finite failure software reliability growth models [8]. According to this model, the expected number of faults detected by time t, called the *mean value function*, $m(t)$ is of the form,

$$m(t) = a * c(t), \tag{1}$$

where a is the expected number of faults in the software (before testing/debugging begins), and $c(t)$ is the coverage function. The proportionality of $m(t)$ and $c(t)$ has been independently confirmed by Malaiya [36]. Table 1 shows the four coverage functions that the ENHPP model used

Coverage Function	Parameters	$m(t)$	Failure Occurrence Rate per fault
Exponential	a, g	$a(1-e^{-gt})$	Constant
Weibull	a, g, γ	$a(1-e^{-gt^{\gamma}})$	Increasing/Decreasing
S-shaped	a, g	$a[1-(1+gt)e^{-gt}]$	Increasing
Log-logistic	a, λ, κ	$a\frac{(\lambda t)^{\kappa}}{1+(\lambda t)^{\kappa}}$	Inverse Bath Tub

Table 1: Coverage functions and parameters estimated

by SREPT provides by default to reflect four types of failure occurrence rates per fault - constant, increasing/decreasing (based on the value of one of the model parameters), increasing, and inverse bath tub. The log-logistic model [37] was developed to deal with failure data sets that cannot be modeled by a constant, or monotonic increasing or monotonic decreasing failure occurrence rate per fault. The nature of the failure occurrence rates per fault of the four coverage functions are plotted in Figure 2. Given the sequence of times between failures $(t_1, t_2, \ldots)$ the object is to estimate the parameters of the chosen model(s). Commonly, one of these parameters is the expected number faults in the program (before testing/debugging begins), denoted by a, as in Equation 1.

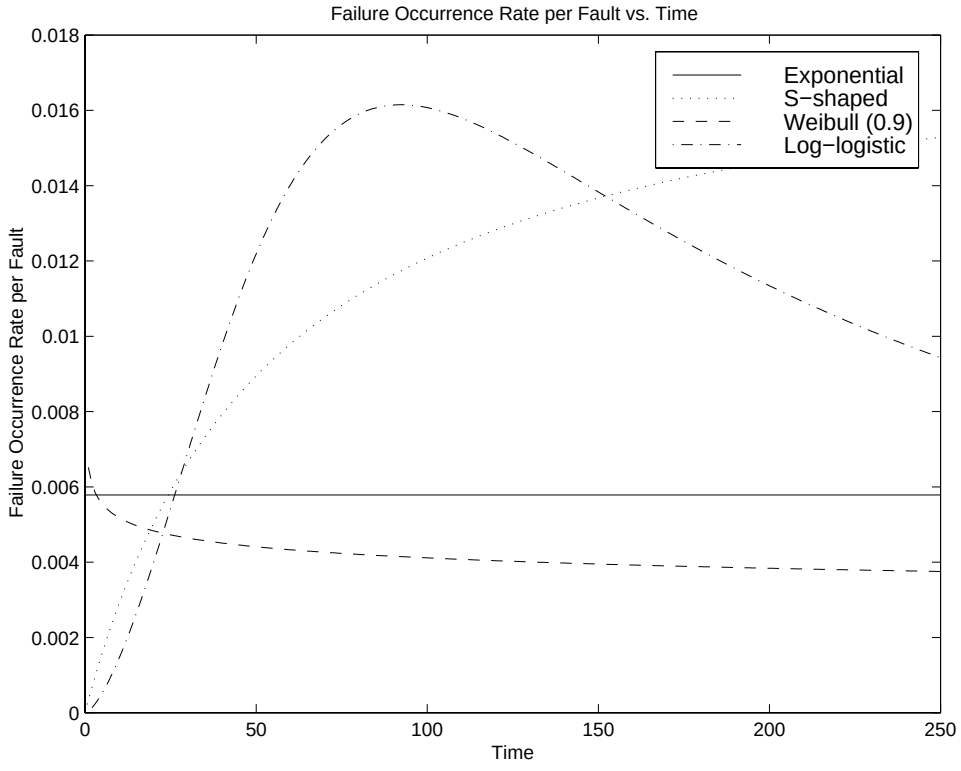

Figure 2: Failure occurrence rates per fault

By explicitly incorporating the time-varying test-coverage function in its analytical formulation (Equation 1), the ENHPP model is capable of handling any general coverage function and provides a methodology to integrate test coverage measurements into the black-box modeling approach. Therefore, when **test coverage** information is available from the testing phase, the user may supply this as coverage measurements at different points in time during the testing phase, or as a time-function. This approach, combining test coverage and interfailure times data is shown in Figure 3.

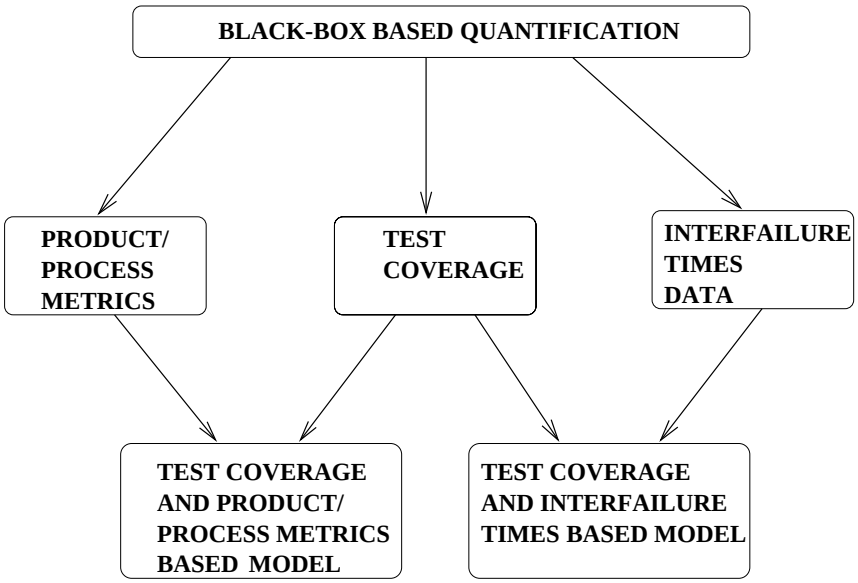

Figure 3: Black-Box Quantification

The framework of the ENHPP model may also be used to combine the estimate of the total number of faults obtained before the testing phase based on product/process metrics (parameter a), with the coverage information obtained during the testing phase ($c(t)$). This is also shown in Figure 3.

The ENHPP model can also use interfailure times from the testing phase to obtain *release times* (optimal time to stop testing) for the software on the basis of a specified **release criteria** [48]. Release criteria could be of the following types -

- Number of remaining faults - In this case, the release time is when a fraction ρ of all detectable faults has been removed.
- Failure intensity requirements - The criterion based on failure intensity suggests that the software should be released when the failure intensity measured at the end of the development test phase reaches a specified value λ_f.

- Reliability requirements - This criteria could be used to specify that the required conditional reliability in the operational phase is, say R_r at time t_0 after product release.

- Cost requirements - From a knowledge of the expected cost of removing a fault during testing, the expected cost of removing a fault during operation, and the expected cost of software testing per unit time, the total cost can be computed. The release time is obtained by determining the time that minimizes this total cost.

- Availability requirements - Assuming that the mean time to repair is the time needed to reboot and that there is no reliability growth, the release time can be estimated based on an operational availability requirement.

The ENHPP-based techniques described above assume instantaneous debugging of the faults detected during testing. This is obviously unrealistic and leads to optimistic estimates. This drawback can be remedied if the **debugging rate** (repair rate) can be specified along with the **failure intensity**. SREPT offers two approaches to analyze explicit fault removal - *state-space method* [27] and *discrete-event simulation* [30].

In the state-space approach, SREPT takes the failure occurrence and repair rates to construct a non-homogeneous continuous time Markov chain (NHCTMC) [7]. The solution of the NHCTMC is obtained using the SHARPE [21] modeling tool that is built into SREPT. Figure 4 shows the NHCTMC for the case of constant repair rate μ. The state of the NHCTMC is represented as a 2-tuple (i, j) where i is the number of faults repaired, and j is the number of faults detected but not yet repaired. Initially, no faults have been detected or repaired. This is represented by the state (0,0). Failures occur at the time-dependent rate $\lambda(t)$ (hence the need to model as a "non-homogeneous" continuous time Markov chain). When the first failure occurs, there is one detected fault which has not been repaired yet. State (0,1) represents this. This fault is repaired at the constant rate μ. If the fault is repaired before the next failure occurs, the next state is (1,0), otherwise the next state will be (0,2), representing two detected but unrepaired faults. Since the NHCTMC has an infinite state space, it is necessary to truncate it for numerical solution. So we truncate it at (i, j) where $i + j = A$. A may be referred to as the "truncation level" used. When applying this methodology, one can either use a high truncation level, or verify by trying out a slightly higher value for A that the truncation level used is adequate (i.e., the two A values used give consistent solutions). The solution of the NHCTMC using SHARPE will enable estimates of *failure intensity, number of faults remaining*, and *conditional reliability* to be obtained.

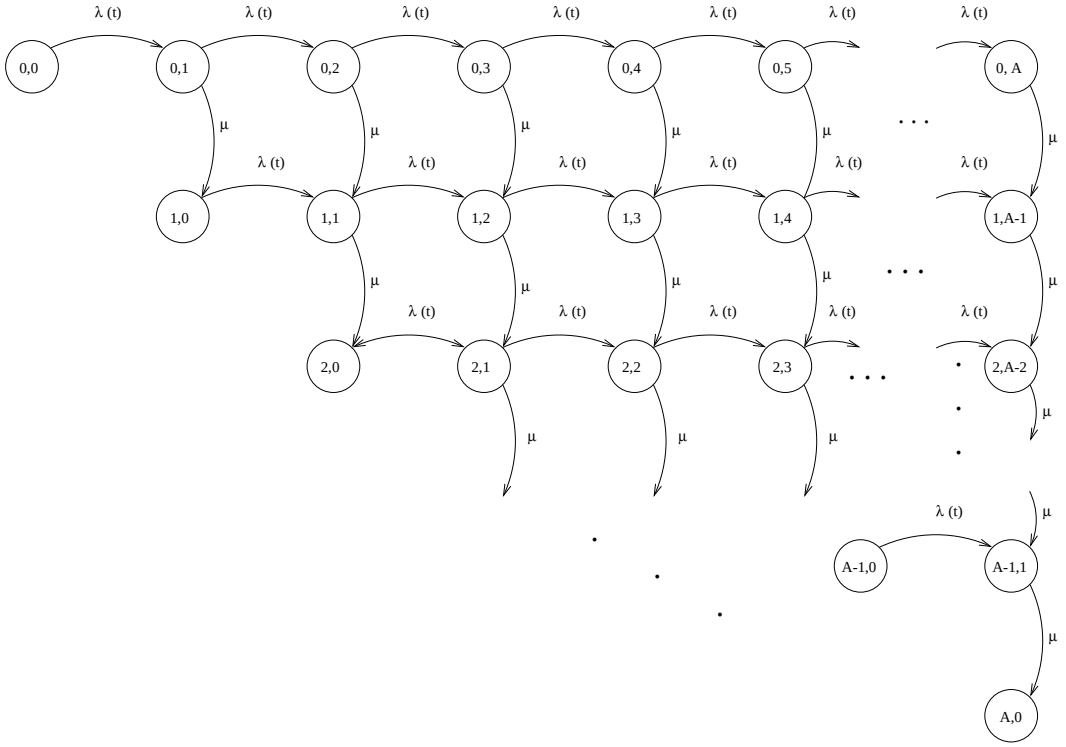

Figure 4: NHCTMC - Constant fault removal rate

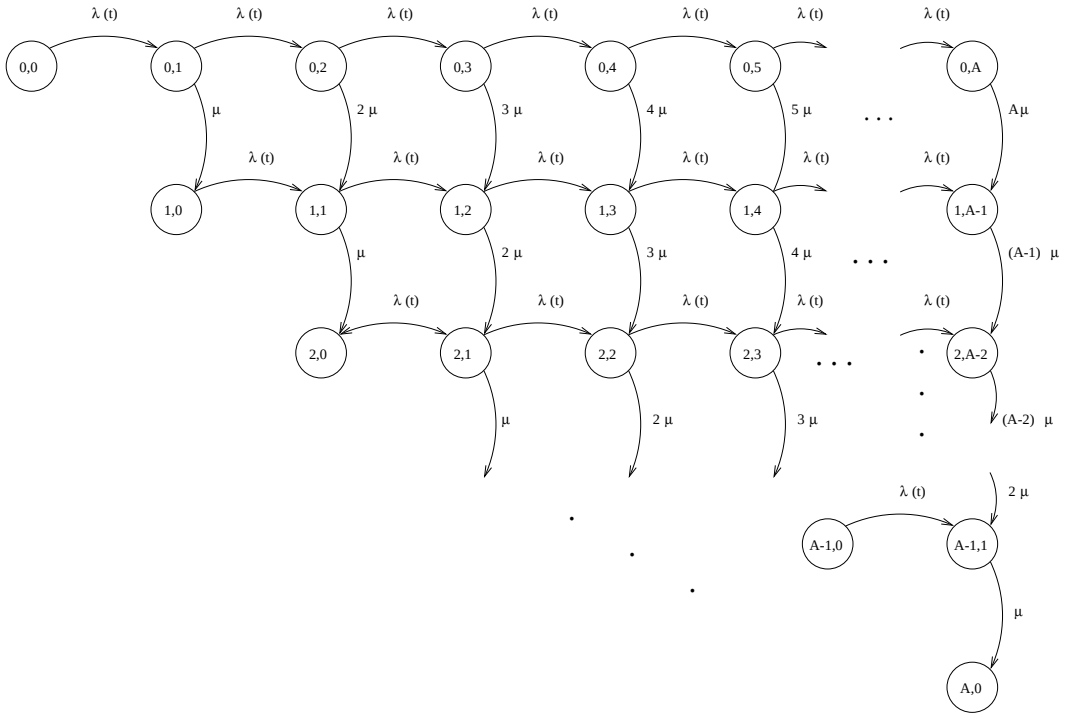

Figure 5: NHCTMC - Fault dependent removal rate

By specifying different repair rates, various repair strategies can be studied [27]. For example, repair rates could be

- constant (say μ) - NHCTMC shown in Figure 4.
- time-dependent (say $\mu(t)$)
- dependent on the number of faults pending for removal - NHCTMC shown in Figure 5.

Other scenarios that can be analyzed using the state-space approach include delayed fault removals. Delayed fault removals can be of two types:

- *Deferred fault removal* - fault removal can be delayed till ϕ faults are detected and are pending for removal.
- *Delayed fault removal* - there is a time delay before the fault removal process begins after the detection of a fault. Delayed fault removal can be incorporated into the NHCTMC using a phase type distribution [7]. The corresponding NHCTMC is shown in Figure 6. If the mean time in phase 1 is given by $1/\mu_1$, and the mean time in phase 2 by $1/\mu_2$, the mean repair time is given by,

$$\frac{1}{\mu} = \frac{1}{\mu_1} + \frac{1}{\mu_2} \tag{2}$$

  The state (i, j, d) indicates that i faults have been removed, j have been detected and are queued for removal, and the d denotes the intermediate phase of repair.

The discrete-event simulation (DES) approach offers distinct advantages over the state-space approach -

- DES accommodates any type of stochastic process. The state-space method is restricted to Markov chains with finite number of states, since a numerical solution of the Markov chain is required.
- Simulation approaches are faster for large NHCTMC's that could be very time-consuming to solve using numerical methods.
- The number of states in the Markov chain increases dramatically with an increase in the truncation level used (A in the discussion of the state-space method). SHARPE imposes restrictions on the number of states in the Markov chain. DES imposes no such restrictions. But DES requires a large number of runs in order to obtain tight confidence intervals.

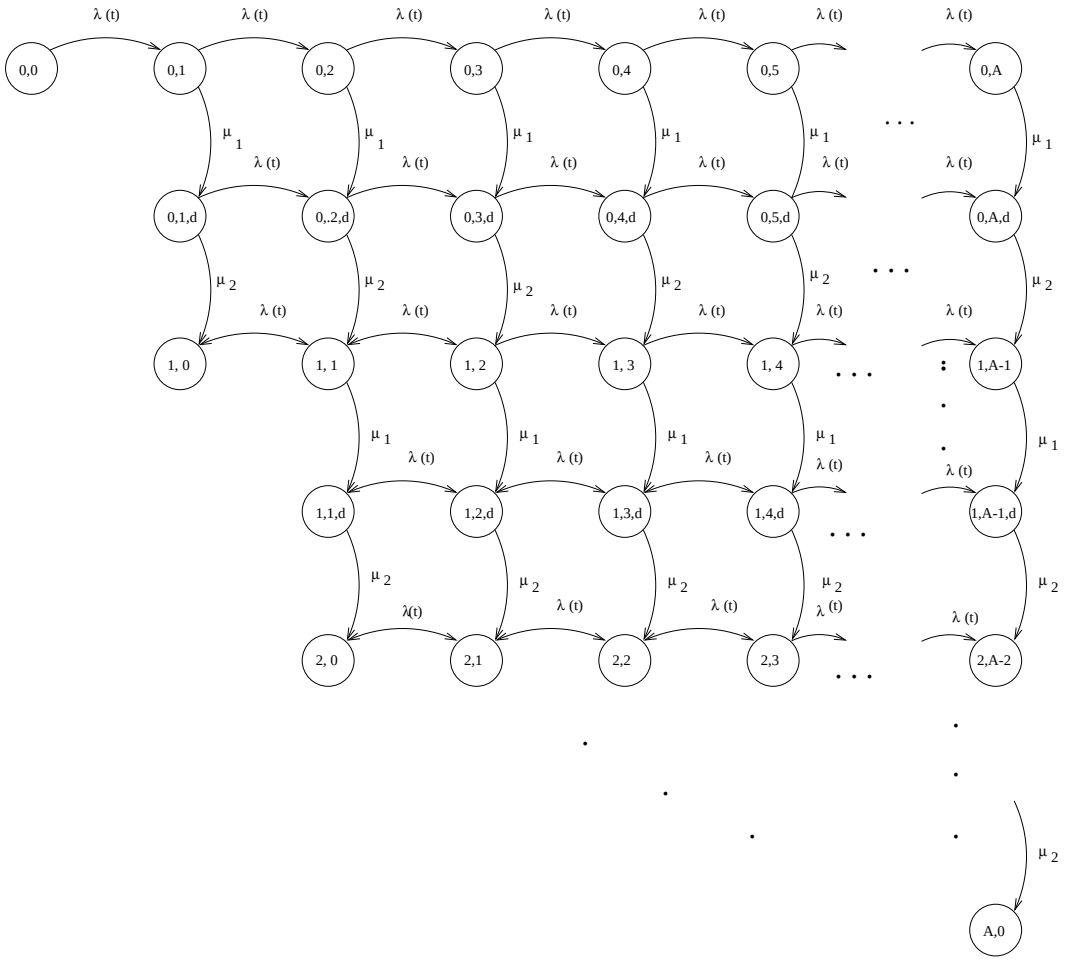

Figure 6: NHCTMC - Delayed fault removal

The DES approach in SREPT involves a rate-based simulation technique for the arrival process of a NHCTMC [30].

3.2 Architecture Based Approach

The second approach to software reliability supported by SREPT (from Figure 1) is the architecture-based approach [26]. Architecture-based approaches use the internal control structure of the software [11] to predict the reliability of software. This assumes added significance in the evaluation of the reliability of modern software systems which are not monolithic entities, but are likely to be made up of several modules distributed globally. SREPT allows prediction of the reliability of software based on :

- **Architecture of the software** : This is a specification of the manner in which the different modules in the software interact, and is given by the intermodular transition probabilities, or in a very broad sense, the operational profile of the software [23]. The architecture of the application can be modeled as a DTMC (Discrete Time Markov Chain), CTMC (Continuous Time Markov Chain), SMP (Semi-Markov Process), SPN (Stochastic Petri Net) or a DAG (Directed Acyclic Graph) [7] [21]. The state of the application at any time is given by the module executing at that time, and the state transitions represent the transfer of control among the modules. DTMC, CTMC, SMP and SPN can be further classified into irreducible and absorbing categories, where the former represents an infinitely running application, and the latter a terminating one. The irreducible DTMC, CTMC, SMP and SPN can be analyzed to compute the state probabilities - the probability that a given module is executing at a particular time, or in the steady state. The absorbing DTMC, CTMC, SMP, SPN and DAG can be analyzed to give performance measures like the expected number of visits to a component during a single execution of the application, the average time spent in a component during a single execution, and the time to completion of the application. DTMC, CTMC, SMP and SPN can be used to model sequential applications while concurrent applications can be modeled using SPN and DAG.

- **Failure behavior** : This specifies the failure behavior of the modules and that of the interfaces between the modules, in terms of the probability of failure (or reliability), or failure rate (constant/time-dependent). Transitions among the modules could either be instantaneous or there could be an overhead in terms of time. In either case, the interfaces could be perfect

or subject to failure, and the failure behavior of the interfaces can also be described by the reliabilities or constant/time-dependent failure rates.

The architecture of the software provides its *performance model*. It can be used to identify performance bottlenecks, perform platform change analysis, analysis in the event of porting the software from a sequential platform to a distributed one, and so on. Various performance measures such as the time to completion of the application, can be computed from the performance model. The architecture of the software can be combined with the failure behavior of the modules and that of the interfaces into a *composite model* which can then be analyzed to predict the reliability of the software. Another approach is to solve the architecture model and superimpose the failure behavior of the modules and the interfaces on to the solution to predict reliability. This is referred to as the *hierarchical model*. These two methods are depicted in Figure 7.

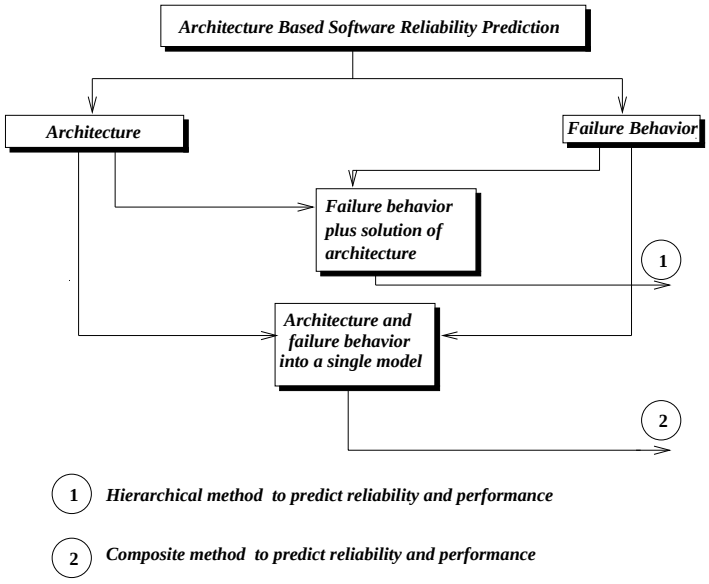

Figure 7: Specification and Analysis Methods of Structural Models

SREPT also supports architecture-based software reliability evaluation using *discrete-event simulation* [31] as an alternative to analytical models. Discrete-event simulation can capture detailed system structure, with the advantage that it is not subject to state-space explosion problems like the analytical methods described before.

In summary, the advantages of the architecture-based approach to software reliability prediction include,

- Performance measures

- Evaluation of design alternatives
- Decision-making with respect to in-house development of software components as opposed to using *commercial off-the-shelf* components
- Identification of reliability and performance bottlenecks

Illustrations of the application of the architecture-based approach can be found in [32] [31].

4 Illustration

This section illustrates the use of SREPT in software reliability estimation and prediction.

4.1 Using *Software Product/Process Metrics*

Figure 8 shows a snapshot of the interface SREPT provides for estimating the number of faults in the software based on the *lines of code* metric. As can be seen, the user can provide information based on a similar previous project. This information is used to obtain the fault density, which is then used to estimate the number of faults present in the modules of the current software product. Figure 8 uses the modules of SHARPE [21] as an example. This approach can be used before actually spending time testing the software. Product/process metrics give a "feel" for the fault content of the software based on the experience of previous software projects, and can help suggest where it would be most beneficial to direct testing effort, so as to maximize the return on investments.

4.2 Using the *ENHPP Engine*

This section gives some examples of the use of the ENHPP engine of SREPT in estimating software reliability and *release time* for the software. Figure 9 shows a snapshot of the tool if the ENHPP engine is chosen. Among the details of the tool revealed in this snapshot are - a means of specifying the type of input data (which can be one of *interfailure times data only, interfailure times and coverage data, or estimated # faults and coverage data*). In the snapshot, it has been specified that the input to the ENHPP engine will consist of *interfailure times data only.* This data should be in a file that should be specified in a textfield. The file chosen in the current example is *"test.dat"*. The user can then choose one of the coverage functions offered by the ENHPP model to estimate the reliability as well as other metrics of interest. The snapshot shows the *Exponential* coverage function (corresponding to the GO-model) selected. When asked to "estimate", the tool then brings

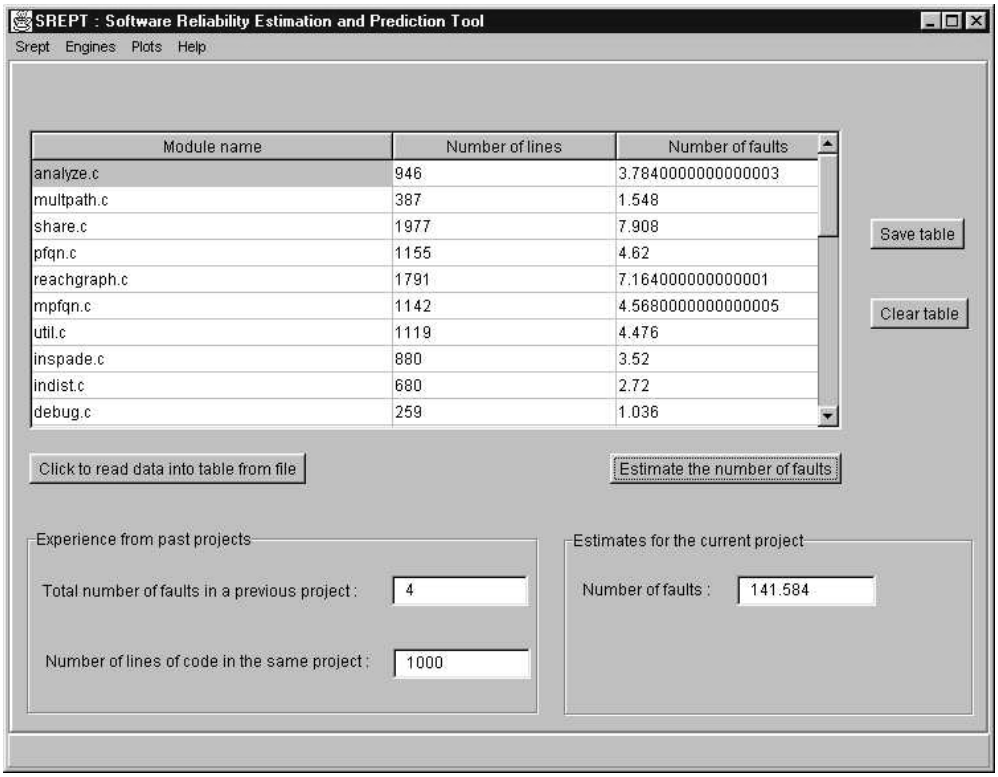

Figure 8: Using *lines of code* metric for the *fault density* approach

up a window with suggested initial values of the parameters of the selected model. The convergence of the numerical routines depends largely on the initial guesses, and hence by changing the initial guesses the user can control it. By using this window to specify values other than those suggested, one can tailor the numerical estimation to suit one's needs. This is especially useful if the user wants to suggest some other initial values based on experience from past software development projects. Figure 10 shows the results of the estimation. The expected number of faults in the software, the estimated model parameters, and the mean value function are displayed.

Plots of metrics such as the *mean value function*, *failure intensity*, *number of faults remaining*, *conditional reliability*, and *estimated coverage* can now be obtained from the solution of the model above. Figure 11 shows a snapshot of the plot of the mean value function. The plot also contains the actual number of faults detected (the irregular plot) that is obtained from the interfailure times data. Figure 12 shows the conditional reliability starting at the time the last fault was detected (which may be the end of the testing phase for the software, if the interfailure times data supplied cover the entire testing phase). The mean value function (that represents the expected number of faults detected by some time t) shows that in the beginning no faults have been detected, and as time progresses more and more faults are expected to be detected. The conditional reliability curve

shows the probability of fault-free operation of the software at a time t beyond the end of testing.

Figure 9 also shows a *button* that can be used to launch the estimation of a release time for the software. This action brings up a window that can be used to select the *release criterion* to be used in estimating a release time. Figure 13 shows this window in which the criterion, *Faults remaining*, has been selected.

Similar plots can be drawn and release times obtained for every other chosen coverage function. SREPT also allows the user to let the ENHPP engine determine the best model for the given interfailure times data. This is done on the basis of a *goodness-of-fit*, *u-plot* and *y-plot* [15]. Figure 14 shows an example of a u-plot.

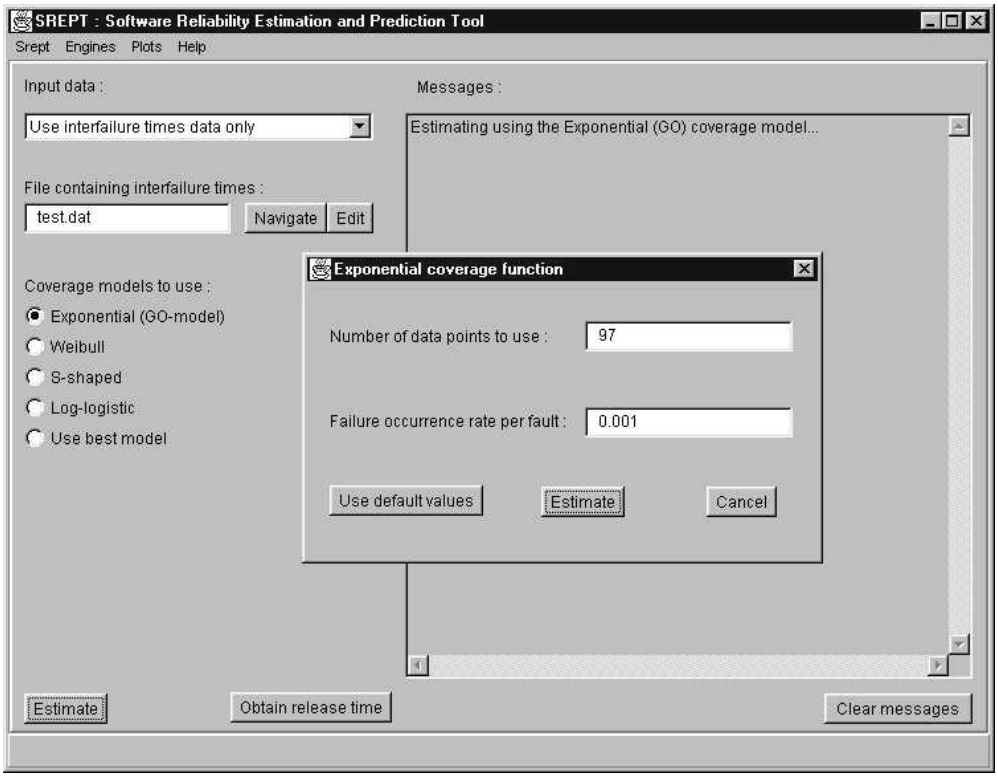

Figure 9: Choosing the ENHPP engine of SREPT

4.3 Specifying *Finite Repair Times*

SREPT can also be used to evaluate the effect of finite repair times (unlike the ENHPP models, that assume instantaneous and perfect repair). Figure 15 shows the interface for this module. The user should specify the truncation level to be used, the failure intensity and the fault removal (repair) rate. The fault removal process can be specified as *deferred* and/or *delayed*. Choosing a "delayed"

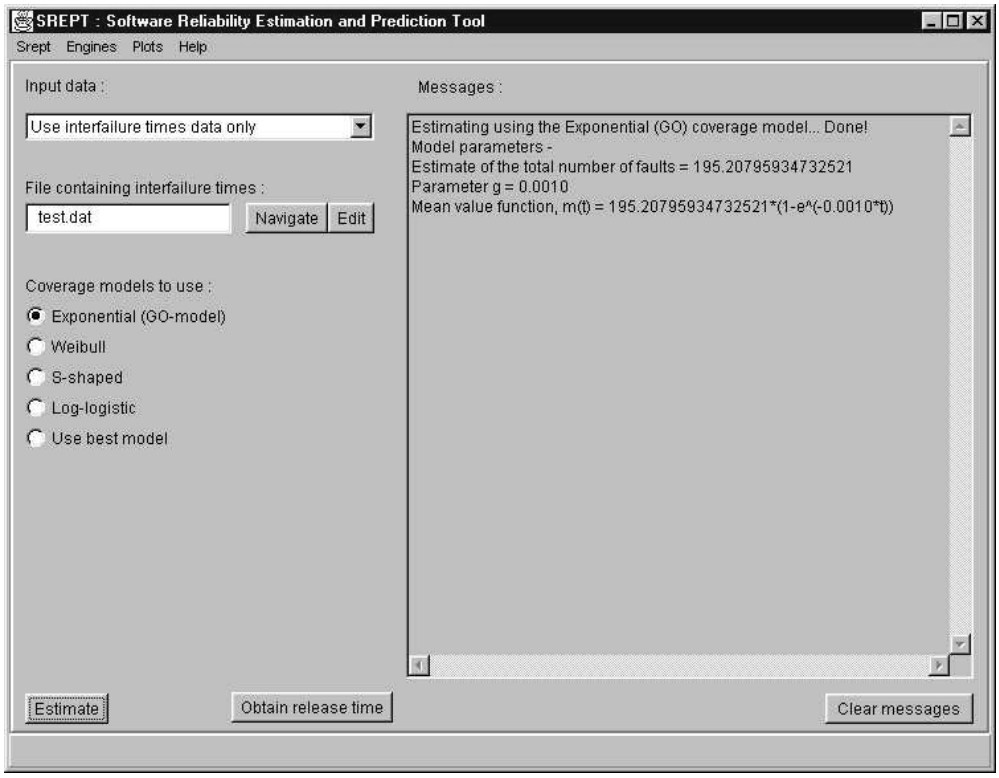

Figure 10: Results from the ENHPP engine of SREPT

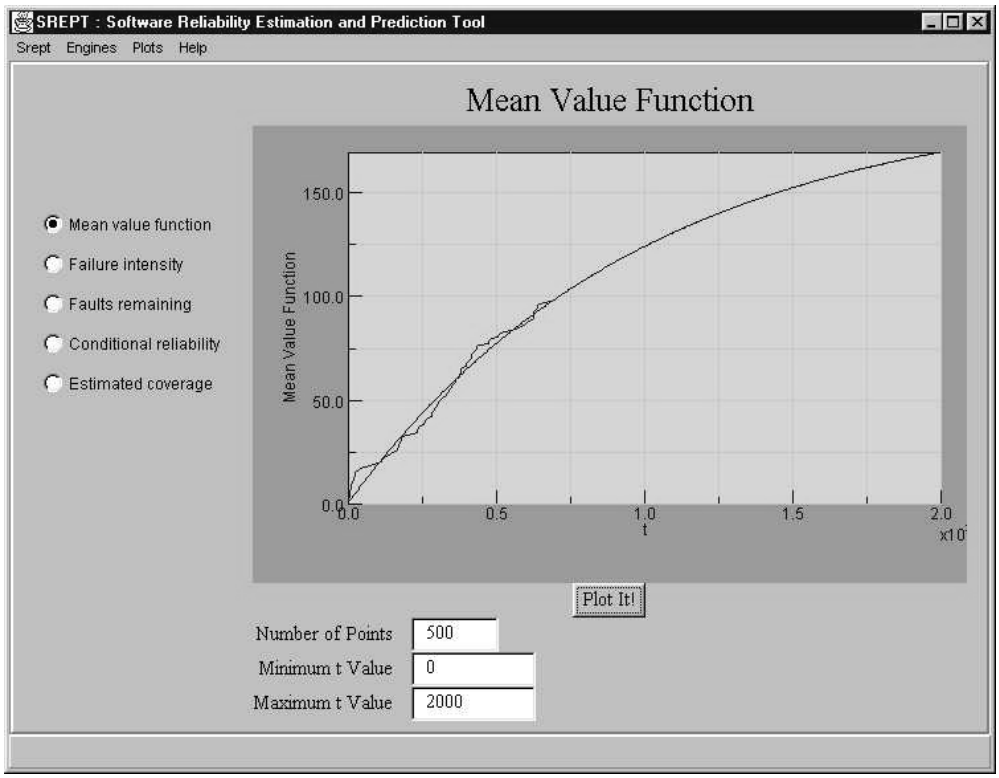

Figure 11: Plot of the *mean value function* from the ENHPP engine

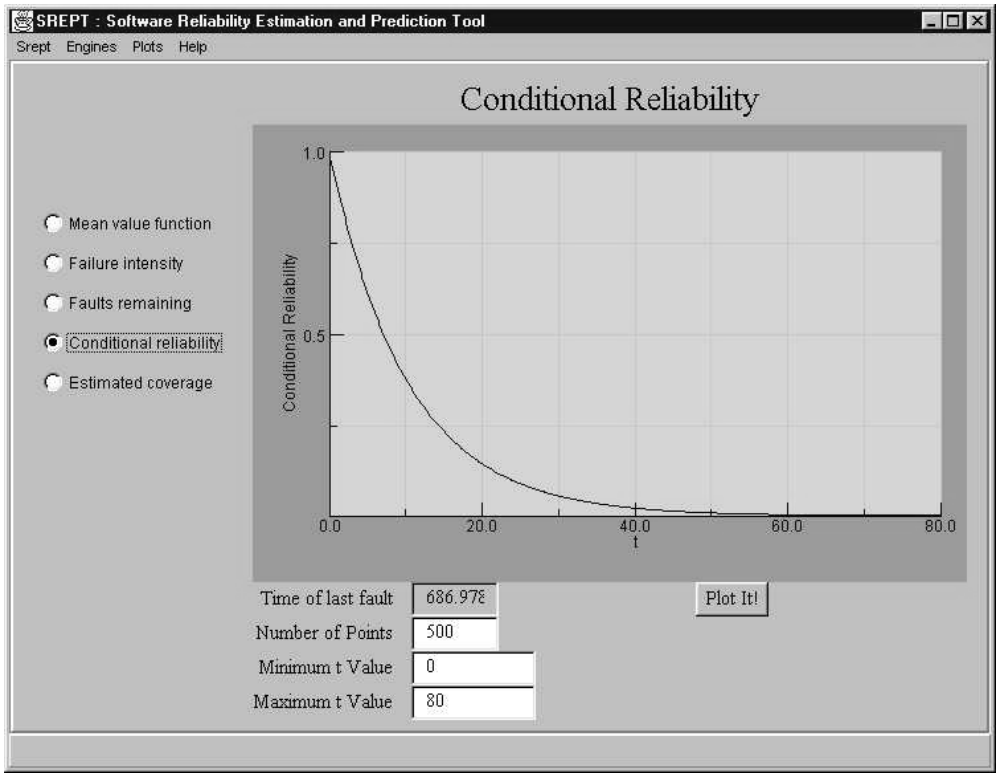

Figure 12: Plots of the *conditional reliability* from the ENHPP engine

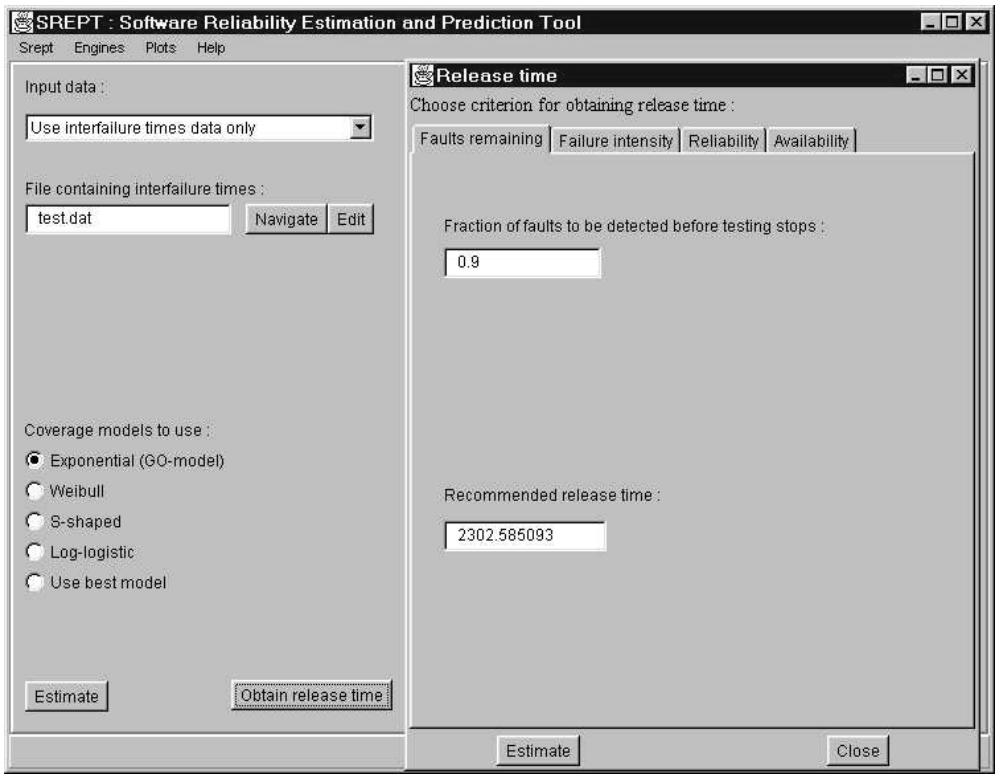

Figure 13: Release time based on *Faults remaining* criterion

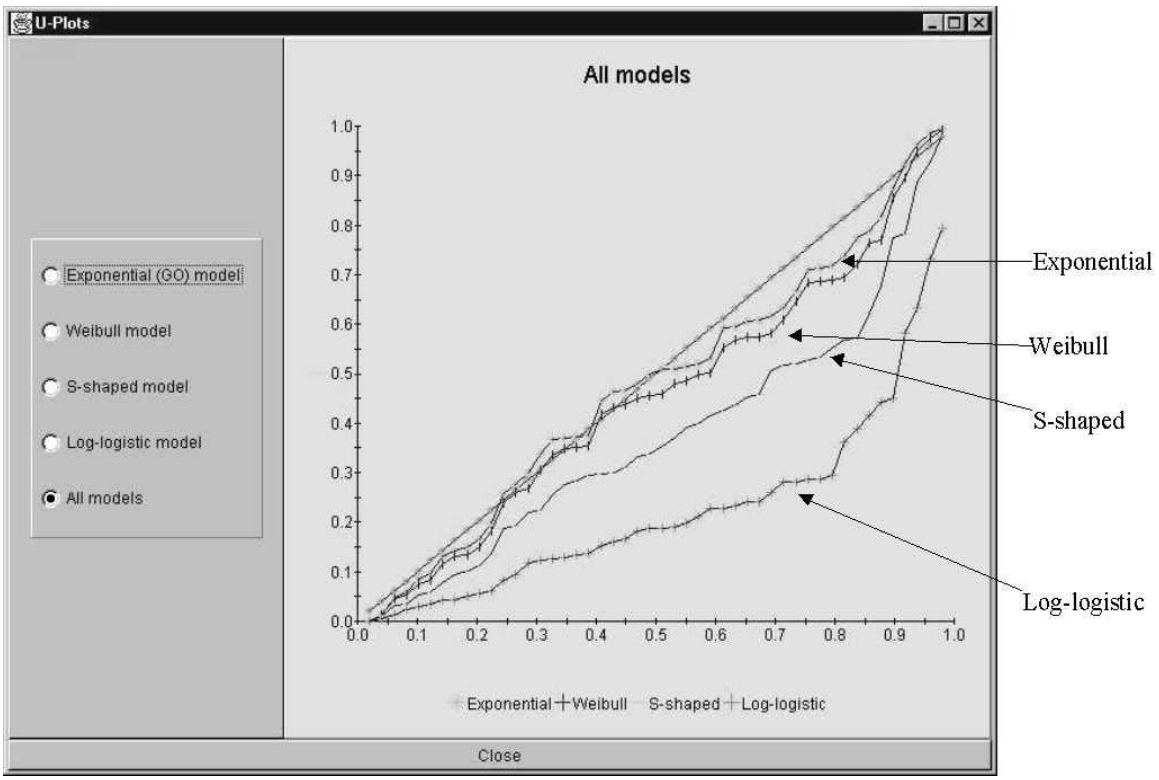

Figure 14: Example of a U-plot

process allows fault removal rates for more than one debugging phase to be specified. When asked to "solve", a configuration window is brought up to enable specification of the time up to which the underlying NHCTMC model that will be constructed is to be solved. The user can also specify at this point, a different time interval that SHARPE should use to approximate the solution of a NHCTMC than the one suggested by default. The use of smaller time intervals results in more accurate solutions at the cost of greater computation time.

The results of the solution are presented in the form of a table, as shown in Figure 16. The table displays the *time, expected number of faults detected, expected number of faults removed, perceived failure intensity*, and *actual failure intensity* (that is obtained by considering the finite repair times). Figure 17 shows the plot of the perceived and actual failure intensities for an example. A plot of the *number of faults remaining* can also be obtained.

Figure 15 shows that the user can also view the SHARPE code generated by SREPT by clicking the appropriate button.

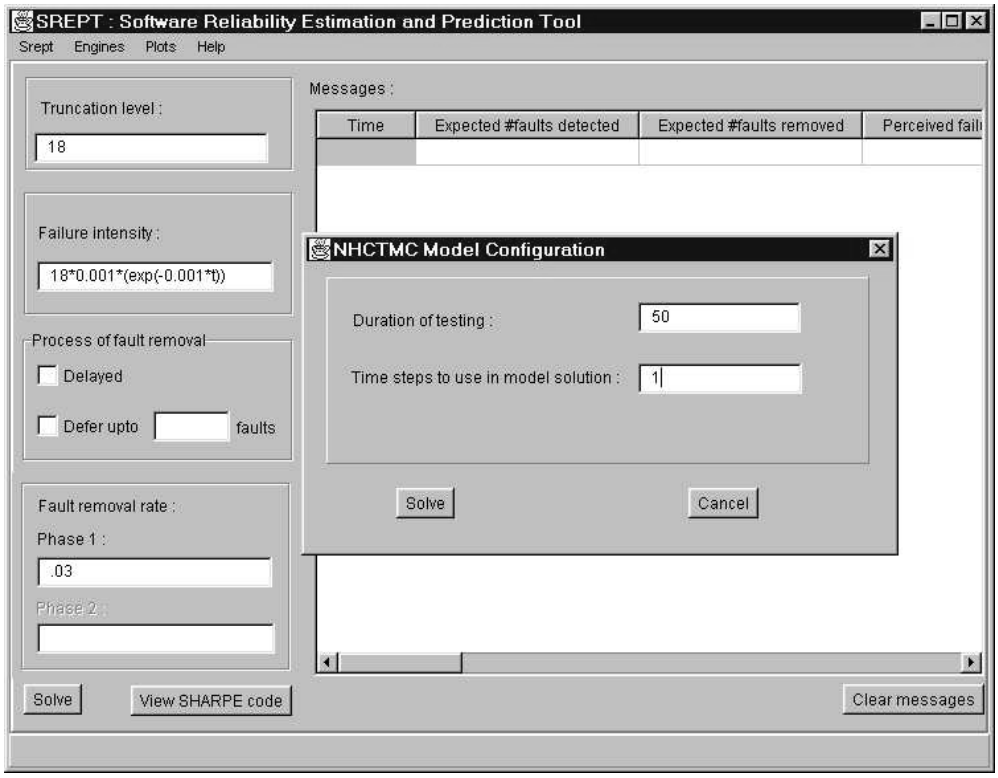

Figure 15: Interface for specifying finite repair times

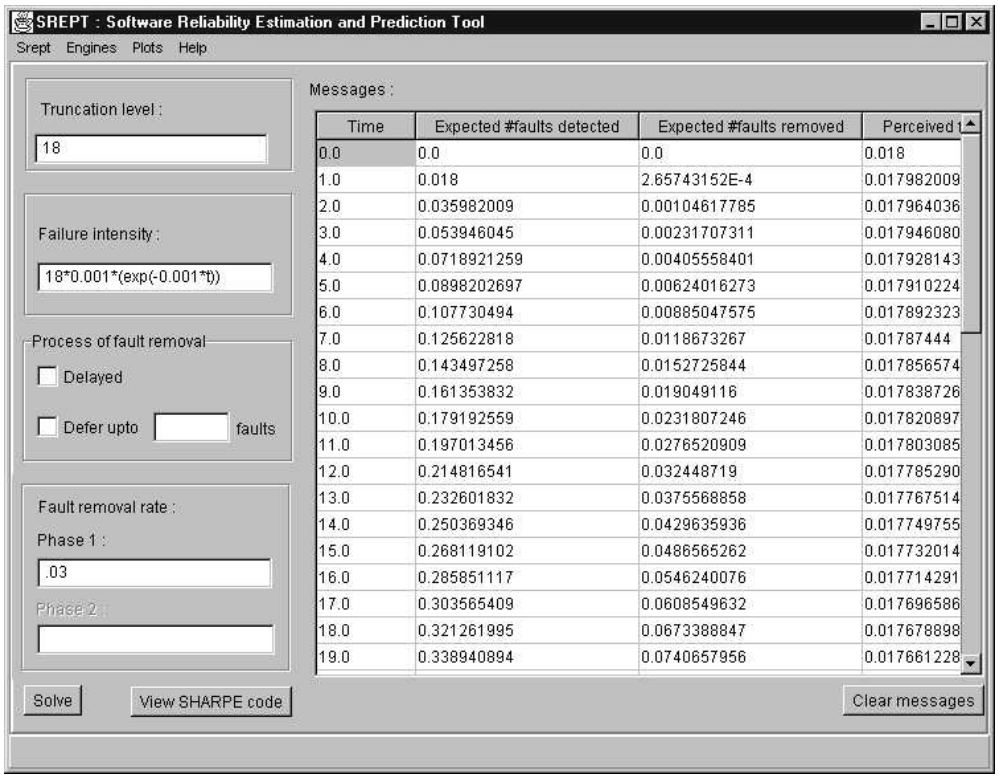

Figure 16: Results from the solution of the NHCTMC model

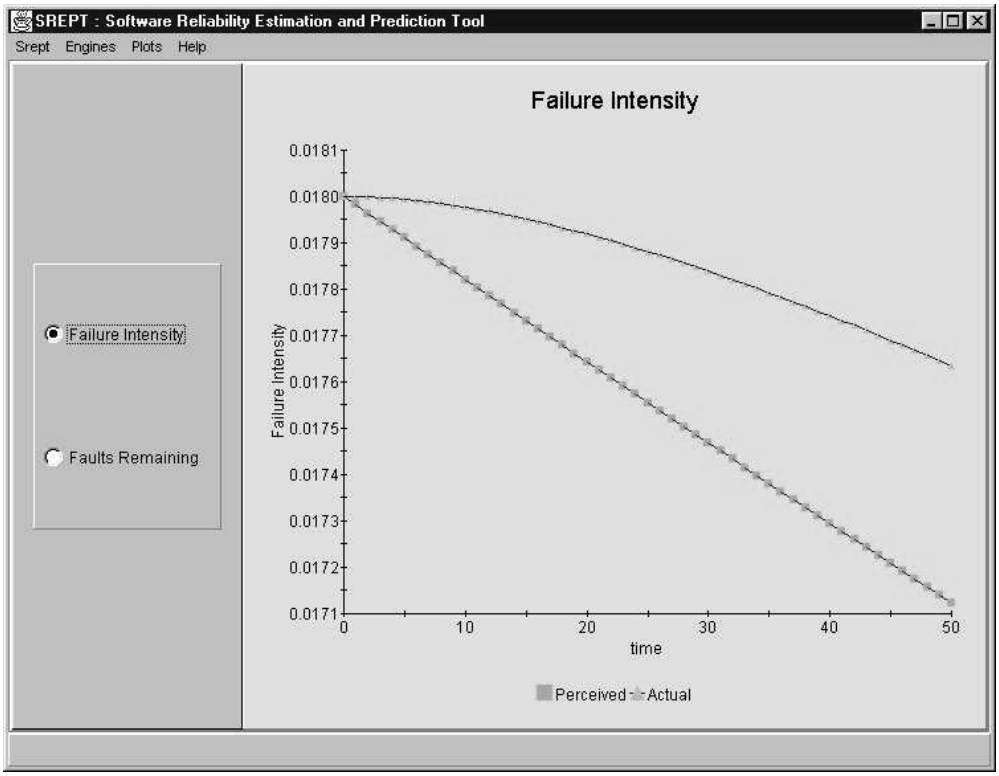

Figure 17: Plot of the perceived and actual failure intensity

4.4 Using the Architecture-Based Approach

The interface for the architecture-based approach is shown in Figure 18. In the figure, a *hierarchical model* is chosen, with the architecture of the software specified by an absorbing DTMC and the failure behavior of the components by their reliabilities. The mean execution times for each module in the software (measured from experiments) can also be specified using the appropriate interface. As and when specifications for the model are made, the predictions obtainable from the current selection are displayed.

5 Expected Impact of SREPT

Figure 19 shows SREPT's position in the life-cycle of a software product. As can be seen from the figure, SREPT offers several techniques that can be used at various stages in the software life-cycle, according to a *waterfall model* [28]. Thus it makes it possible to monitor the quality of the entire software development process under a unified framework for software reliability estimation and prediction. We expect that this aspect along with its ability to perform architecture-based predictions will enable SREPT to have a powerful impact in the area of software reliability estimation and

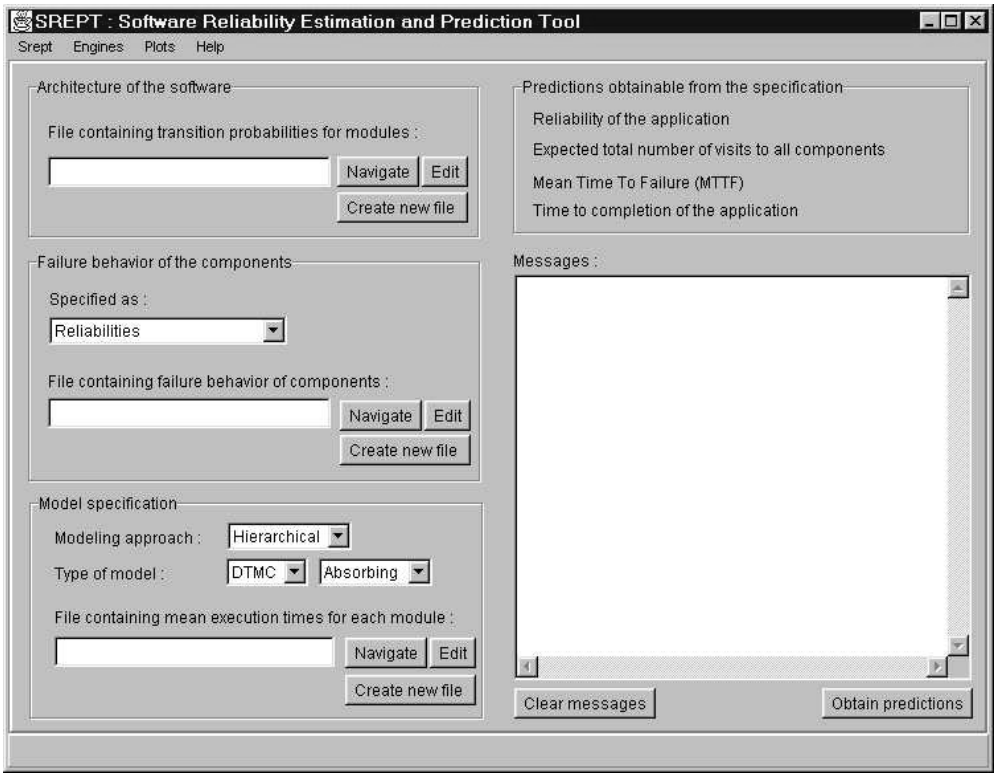

Figure 18: Interface for the architecture-based approaches

prediction.

6 Conclusion

In this paper we presented the high-level design of a tool offering a unified framework for software reliability estimation that can be used to assist in evaluating the quality of a software product through its development process, right from the architectural phase all the way up to the operational phase. This is because the tool implements several software reliability techniques including software product/process metrics-based techniques used in the pre-test phase, interfailure times-based techniques used during the testing phase, and architecture-based techniques that can be used at all stages in the software's life-cycle. Architecture-based techniques are being implemented in a tool in a systematic manner for the first time. SREPT also has the ability to suggest release times for software based on release criteria, and has techniques that incorporate finite repair times while evaluating software reliability. The tool is expected to have a widespread impact because of its applicability at multiple phases in the software life-cycle, and the incorporation of several techniques in a systematic, user-friendly form in a GUI-based environment.

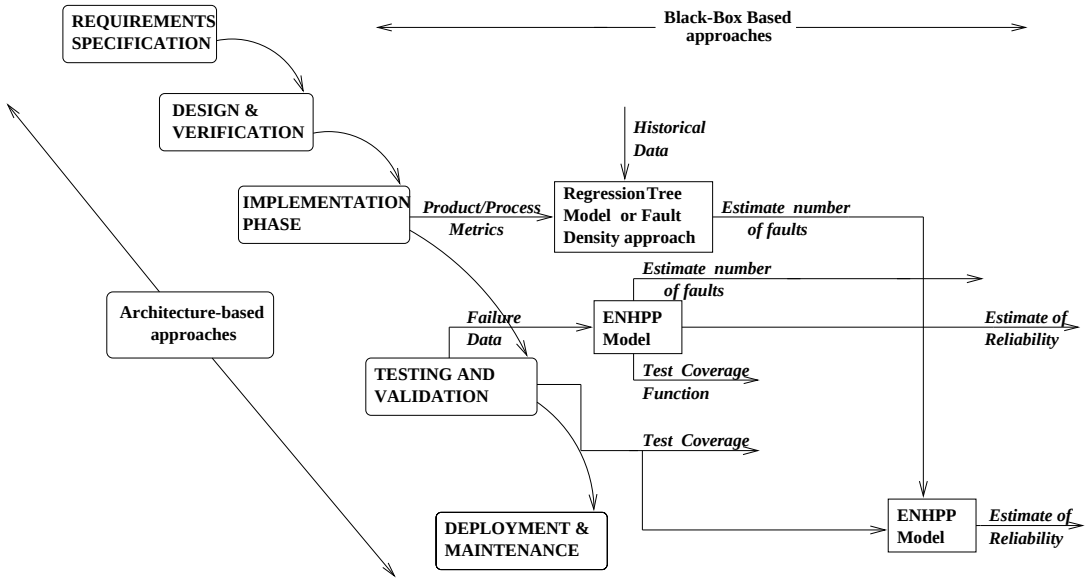

Figure 19: Application of SREPT at Various Phases of Software Evolution

Acknowledgement

This paper is an extended version of a paper [33] that appeared in the proceedings of the TOOLS '98 conference, published in *Lecture Notes in Computer Science 1469* by Springer-Verlag.

References

[1] ANSI/IEEE, "Standard Glossary of Software Engineering Terminology", STD-729-1991, ANSI/IEEE, 1991.

[2] S. Gokhale, P.N. Marinos and K.S. Trivedi, "Important Milestones in Software Reliability Modeling", In Proc. of *8th Intl. Conf. on Software Engineering and Knowledge Engineering (SEKE '96)*, pp. 345-352, Lake Tahoe, June 1996.

[3] A.L. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Models for Software Reliability and Other Performance Measures", *IEEE Trans. on Reliability*, Vol.R-28, No.3, pp. 206-211, August 1979.

[4] -, "A Guidebook for Software Reliability Assessment", Rep.RADC-TR-83-176, August 1983.

[5] -, "Software Reliability Modeling and Estimation Techniques", Rep.RADC-TR-82-263, October 1982.

[6] M. Ohba, "Software Reliability Analysis Models", *IBM Journal of Research and Development*, Vol. 28, No. 4, pp. 428-442, July 1984.

[7] K.S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications", Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[8] S. Gokhale, T. Philip, P. N. Marinos and K.S. Trivedi, "Unification of Finite Failure NHPP Models through Test Coverage", In Proc. of *Intl. Symposium on Software Reliability Engineering (ISSRE '96)*, pp. 289-299, White Plains, NY, October 1996.

[9] J.P. Hudepohl, S.J. Aud, T.M. Khoshgoftaar, E.B. Allen and J. Mayrand, "Emerald: Software Metrics and Models on the Desktop", *IEEE Software*, pp. 56-60, September 1996.

[10] M.R. Lyu (Editor), *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, NY, 1996.

[11] J.P. Horgan and A.P. Mathur, chapter "Software Testing and Reliability", pp. 531-566, *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, McGraw-Hill, New York, NY, 1996.

[12] W. Farr, chapter "Software reliability Modeling Survey", pp. 71-117, *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, McGraw-Hill, New York, NY, 1996.

[13] M. Lipow, "Number of Faults per Line of Code", *IEEE Trans. on Software Engineering*, Vol. 8, No. 4, pp. 437-439, July 1982.

[14] S. Gokhale and M.R. Lyu, "Regression Tree Modeling for the Prediction of Software Quality", In Proc. of *ISSAT'97*, pp. 31-36, Anaheim, CA, March 1997.

[15] S. Brocklehurst and B. Littlewood, chapter "Techniques for Prediction Analysis and Recalibration", pp. 119-166, *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, McGraw-Hill, New York, NY, 1996.

[16] M.R. Lyu, A.P. Nikora and W.H. Farr, "A Systematic and Comprehensive Tool for Software Reliability Modeling and Measurement", In Proc. of the *23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 648-653, Toulouse, France, June 1993.

[17] M.R. Lyu, and A.P. Nikora, "CASRE- A Computer-Aided Software Reliability Estimation Tool", *CASE 92 Proceedings*, pp. 264-275, Montreal, Canada, July 1992.

[18] A.P. Nikora, "CASRE User's Guide", Jet Propulsion Laboratories, August 1993.

[19] W.H. Farr and O. Smith, "Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) User's Guide", NSWCDD TR 84-373, Revision 3, Naval Surface Warfare Center Dahlgren Division, September 1993.

[20] K. Kanoun, M. Kaaniche, J.C. Laprie and S. Metge, "SoRel: A Tool for Reliability Growth Analysis and Prediction from Statistical Failure Data", Center National de la Recherche Scientifique, LAAS Report 92.468, December 1992. Also in Proc. of the *23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 654-659, Toulouse, France, June 1993.

[21] R.A. Sahner, K.S. Trivedi and A. Puliafito, "Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package", Kluwer Academic Publishers, Boston, 1996.

[22] W.E. Howden, "Functional Program Testing", *IEEE Trans. on Software Engineering*, Vol. 6, No. 2, pp.162-169, March 1980.

[23] J.D. Musa, "Operational Profiles in Software-Reliability Engineering", *IEEE Software*, Vol. 10, No. 2, pp. 14-32, March 1993.

[24] S.R. Dalal and C.L. Mallows, "Some Graphical Aids for Deciding When to Stop Testing Software", *IEEE Journal on Selected Areas in Communications*, Vol.8, No.2, pp. 169-175, February 1990.

[25] S. R. Dalal and A. A. McIntosh, "When to Stop Testing for Large Software Systems with Changing Code", *IEEE Trans. on Software Engineering*, Vol.20, No.4, pp. 318-323, April 1994.

[26] S. Gokhale and K.S. Trivedi, "Structure-Based Software Reliability Prediction", In Proc. of *Fifth Intl. Conference on Advanced Computing (ADCOMP '97)*, pp. 447-452, Chennai, India, December 1997.

[27] S. Gokhale, P.N. Marinos, K.S. Trivedi and M.R. Lyu, "Effect of Repair Policies on Software Reliability", In Proc. of *Computer Assurance (COMPASS '97)*, pp. 105-116, Gatheirsburg, Maryland, June 1997.

[28] B. W. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, Vol.21, No. 5, pp. 61-72, 1988.

[29] J.C. Laprie and K. Kanoun, chapter "Software Reliability and System Reliability", pp. 27-69, *Handbook of Software Reliability Engineering, M. R. Lyu, Editor*, McGraw-Hill, New York, NY, 1996.

[30] S. Gokhale, M.R. Lyu and K.S. Trivedi, "Software Reliability Analysis Incorporating Fault Detection and Debugging Activities", In Proc. of *Intl. Symposium on Software Reliability Engineering (ISSRE '98)*, pp. 202-211, Paderborn, Germany, November 1998.

[31] S. Gokhale, M.R. Lyu and K.S. Trivedi, "Reliability Simulation of Component-Based Software Systems", In Proc. of *Intl. Symposium on Software Reliability Engineering (ISSRE '98)*, pp. 192-201, Paderborn, Germany, November 1998.

[32] S. Gokhale, W.E. Wong, K.S. Trivedi and J.R. Horgan, "An Analytical Approach to Architecture-Based Software Reliability Prediction", In Proc. of *Intl. Performance and Dependability Symposium (IPDS '98)*, pp. 13-22, Durham, NC, September 1998.

[33] S. Ramani, S. Gokhale and K.S. Trivedi, "SREPT: Software Reliability Estimation and Prediction Tool", In Proc. of the *10th Intl. Conference on Modelling Techniques and Tools (Tools '98)*, Palma de Mallorca, Spain, published in Lecture Notes in Computer Science 1469, pp. 27-36, Springer-Verlag, September 1998.

[34] J.A. Denton, "ROBUST, An Integrated Software Reliability Tool", *Technical Report*, Colorado State University, 1997.

[35] Y.K. Malaiya and J.A. Denton, "What Do Software Reliability Parameters Represent", In Proc. of *Intl. Symposium on Software Reliability Engineering (ISSRE '97)*, pp. 124-135, Albuquerque, NM, November 1997.

[36] M.N. Li, Y.K. Malaiya and J. Denton, "Estimating the Number of Defects: A Simple and Intuitive Approach", in Fast Abstracts & Industrial Practices, *Intl. Symposium on Software Reliability Engineering (ISSRE '98)*, pp. 307-315, Paderborn, Germany, November 1998.

[37] S. Gokhale and K.S. Trivedi, "Log-Logistic Software Reliability Growth Model", In Proc. of the *Third IEEE Intl. High-Assurance Systems Engineering Symposium (HASE '98)*, pp. 34-41, Washington, DC, November 1998.

[38] P. Liggesmeyer and T. Ackermann, "Applying Reliability Engineering: Empirical Results, Lessons Learned, and Futher Improvements", in Fast Abstracts & Industrial Practices, *Intl.*

Symposium on Software Reliability Engineering (ISSRE '98), pp. 263-271, Paderborn, Germany, November 1998.

[39] -, "M-élopée", Mathix SA - 19, rue du Banquier - 75013, Paris, France, email: mathix@filnet.fr

[40] J.R. Horgan and S. London, "ATAC: A Data Flow Coverage Testing Tool for C", In Proc. of *Second Symposium on Assessment of Quality Software Development Tools*, pp. 2-10, New Orleans, LA, May 1992.

[41] Reliable Software Technologies, *http://www.rstcorp.com*

[42] McCabe & Associates, *http://www.mccabe.com/?file=./prod/vqt.html*

[43] D.L. Parnas, "The Influence of Software Structure on Reliability", In Proc. of *1975 Intl. Conf. on Reliable Software*, pp. 358-362, Los Angeles, CA, April 1975.

[44] M.L. Shooman, "Structural Models for Software Reliability Prediction", In Proc. of *2nd Intl. Conf. on Software Engineering'*, pp. 268-280, San Francisco, CA, October 1976.

[45] S. Krishnamurthy and A.P. Mathur, "On the Estimation of Reliability of a Software System Using Reliabilities of its Components", In Proc. of *Eighth Intl. Symposium on Software Reliability Engineering (ISSRE '97)*, pp. 146-155, Albuquerque, NM, November 1997.

[46] A.J. Perlis, F.G. Sayward and M. Shaw, "Software Metrics: An Analysis and Evaluation", MIT Press, Cambridge, MA, 1981.

[47] W.N. Venables and B.D. Ripley, "Modern Applied Statistics with S-Plus", Springer-Verlag, New York, 1994.

[48] S. Gokhale, "Analysis of Software Reliability and Performance", Ph.D. thesis, Department of Electrical and Computer Engineering, Duke University, 1998.