IV&V ISSUES IN ACHIEVING HIGH RELIABILITY AND SAFETY IN CRITICAL CONTROL SYSTEM SOFTWARE

Allen P. Nikora
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099
Mail Stop 264-805
VOX: (818)393-1104
fax: (818)393-7830
Allen.P.Nikora@jpl.nasa.gov

Norman F. Schneidewind
Naval Postgraduate School
Code Sm/Ss
Monterey, CA 93943
vox: (408)656.2719
fax: (40.S)656 3407
Schneidewind@nps.navy.mil

John C. Munson
Gregory A. Hall
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
vox: (208)885-7789
fax: (208) 8S8.5-9052
{jmunson, ghall}@cs.uidaho.edu

Keywords: Software complexity, software evolution, software functionality, software metrics, software metrics validation, software reliability, software risk assessment

Abstract

Risk analysis and integrated verification and validation are two important elements in a plan for ensuring the safety of critical software systems. We describe an approach we are currently developing for integrating risk analysis, reliability analysis, and metrics analysis, and propose a fault predictor that would integrate the results of these activities. Practical difficulties associated with our approach are also discussed, as are limitations of the proposed predictor. We conclude with a discussion of what has been learned to date, and with suggestions for future work.

1. Introduction

Risk analysis and integrated verification and validation (IV&V) are two important elements in a plan for ensuring the safety of critical software systems. By risk analysis we refer to estimating the probability of the software entering a hazardous state which could lead an undesirable event, such as loss of life or equipment. If the risk is unacceptable, appropriate steps are taken to reduce it to an acceptable level.

According to IEEE Std 610.12, verification ensures that the software produced in each phase is consistent with software produced in the previous phase, while validation ensures that the software is produced in accordance with requirements. Thus IV&V and risk analysis are related: IV&V can be used to reduce risk by employing inspection, testing, safety analysis, reliability analysis, and metrics analysis. Our research is focused on the last three.

In risk reduction, we distinguish between software reliability and software safety. The former is defined as the probability of no failure, while the latter is defined as the probability of no mishap. Furthermore, the purpose of reliability requirements is to make software failure-free and the purpose of safety requirements is to make software mishap-free. Software reliability), and software safety share the goal of designing into the software the reliability and safety that is required to reduce risk to an acceptable level.

As critical systems play increasingly greater roles in our society, it is important to develop approaches that can be used to both reduce risk and increase reliability and safety. With this goal in mind, we are investigating an approach which integrates the following:

- Risk **Analysis**: We are conducting an experiment to see whether requirements risk indicators are statistically related to the software safety risk metrics that we developed and validated for the Space Transportation System (S'1'S) [1];
- **Reliability** Analysis: We are using currently available software reliability models to: a) predict time to next failure and remaining failures for areas of the software that the risk analysis has identified as containing hazards, and b) conversely, using the reliability predictions to identify software that should receive priority attention for risk analysis due to relatively low reliability predictions as indicated by short time to next failure or high remaining failures, and
- **Metrics** Analysis: We are examining the use of metrics as early indicators of reliability. Like risk analysis, the use of metrics is designed to provide early indicators of poor quality so that corrective action can be taken early on while the cost of error correction is minimal. We intend to integrate the use of metrics and reliability by utilizing the metrics validation methodology developed by Schneidewind [2], which is a core part of IEEE Std 1061 Standard for a Software Quality Metrics Methodology. The purpose of the validation will be to identify those metrics which have sufficient association with defect report counts, failure counts and time to next failure - both observed and predicted - to serve as early indicators of reliability.

We describe the details of our approach in Sections 2 through 4.

2. Risk Analysis

2.1 Feasibility of Fault Tree Analysis

Our initial idea was to analyze selected portions of the STS flight control software using Fault Tree Analysis (FTA) to identify safety hazard scenarios, as recommended by the National Research Council (NRC). We were also interested in identifying fault trees at the system level to see whether these could be used to identify parts of the software needing more intensive reliability analysis. Furthermore, we were interested in seeing whether the hazards are related to high failure rates and to high code complexity. However, in detailed discussions with Johnson Space Center (JSC) requirements, safety, and quality engineers and with their counterparts at the flight software developer, Lockheed-Martin (L-M), and with its development and inspection personnel, it was determined that FTA, although not impossible to perform, would be both technically and economically infeasible for existing software, for the following reasons. First of all, requirements are specified with a variety of methods and formats, including control system type diagrams, flow diagrams, state diagrams, and HAL/S code. The requirements are also specified by various organizations, including JSC, Draper Labs, and Rockwell-Downey. Second, many requirements analysts who would be able to explain their requirements development approach have retired from JSC. Also, too many documents and procedures would have to be modified to successfully implement FTA. Finally, the cost of training both JSC and contractor personnel in FTA is outside the scope of our effort.

2.2 Approach

As a result of the discussions with JSC and development contractor personnel, we have decided to use the S'1'S software to conduct an experiment to determine the extent to which requirements risk indicators are statistically related to the software safety risk metrics that we developed and validated for the STS software [1]. Unlike previous work involving S1'S software, this work deals with the front end of the software process where the seeds of major future reliability problems are sown. Incomplete, inconsistent, ambiguous, or otherwise erroneous requirements can cause great problems with software reliability and quality during operation use. In addition, there are risks associated with creating new software or modifying existing software in order to implement a requirements change. The following are typical risk factors:

- Qualitative assessment of change complexity (e.g., per [3, 4]).
- Size of data and code areas affected by a change.
- Number of lines of code affected by a change.
- Effect on CPU performance.
- Area of the program affected by a change.
- Number and types of other requirements affected by a change.
- Whether a software change is on a nominal or off-nominal program path.
- Number of system raid hardware failures that would have to occur before the code implementing the requirement would be executed.
- Operational phases affected.
- New or existing code that is affected by a change.
- Possible conflicts among requirements changes.
- Effort required to implement a change.
- Effort required to verify and validate the correctness of a change.

- Software tool creation/modification needed to implement a change

The software safety aad risk metrics described in [1] address the two safety goals described below. Defining our safety goal as the reduction of failures that would cause loss of life, loss of mission, or abort of mission to an acceptable level of risk [5], then for software to be ready to deploy, after having been tested for total time t_t, we must satisfy the following criteria:

1) predicted remaining failures $R(t_t)<R_c$, where R_c is a specified critical value, and (1)

2) predicted time to next failure $T_F(t_t)>t_m$, (2) where t_m is mission duration

Both criteria are needed because they relate to two different but related aspects of risk: 1) residual faults that may be in the software during a mission and 2) a failure occurring before the mission is complete. Since the STS is tested and operated continuously, t_t, $T_F(q)$, and t_m are measured in execution time. As with any software safety assurance methodology, we cannot guarantee safety. Rather, with these criteria, we seek to reduce the risk of deploying the software to an acceptable level.

2.2.1 Remaining Failures Criterion

Assuming that the faults causing failures rac removed (as is the case for the S1'S), *criterion 1* specifies that residual failures and faults must be reduced to a level where the risk of operating the software is acceptable. We suggest $R_c=1$, or having the expected value of the *remaining failures* be less than one before deploying the software. If we predict $R(t_t)<R_c$, the mission could begin at t_t. However, if $R(t_t)>R_c$, we would test for a total time $t_t'>t_t$ until $R(t_t')<R_c$, assuming that we will experience more failures and correct more faults so that the *remaining failures* will be reduced by $R(t_t)-R(t_t')$. If a developer has insufficient resources or is otherwise unable to satisfy the criterion, the risk of deploying the software prematurely should be assessed (see Section 2.3). In both cases *criterion 2)* must also be satisfied for the mission to begin.

One way to specify R_c is by failure severity level (e.g., *severity level 1* for life threatening failures). Another, more demanding, method, is to specify that R_c represents *all* severity levels. For example, $R(t_t)<1$ would mean that $R(t_t)$ must be less than one failure, *independent* of severity level.

2.2.2 Time to Next Failure Criterion

Criterion 2 states that the software must survive for a time greater than the duration of the mission. If we predict $T_F(t_t)>t_m$, the mission could begin at t_t. However, if $T_F(t_t)<t_m$, we would continue to test for a total time $t_t''>t_t$ until we are able to predict $T_F(t_t'')>t_m$, assuming that we will experience more failures and correct more faults so that the time to next failure will be increased by the quantity $T_F(t_t'')-T_F(t_t)$. Again, if it is infeasible for the developer to satisfy the criterion for lack of resources m failure to achieve test objectives, the risk of deploying the software prematurely should be assessed (see Section 2.3). In both cases *criterion 1)* must also be satisfied for the mission to begin. If neither criterion is satisfied, we test for a time which is the greater of t_t' or t_t''.

2.3 Risk Assessment

The amount of total test execution time t_t can be considered a measure of the software's maturity, particularly for systems like the S1'S where the software is subjected to continuous and rigorous testing for several years in multiple facilities (c. g, by Lockheed-Martin in Houston, by NASA in Houston for astronaut training, and by NASA al Cape Canaveral), using a variety of operational and training scenarios. If we view t_t as an input to a risk reduction process, and $R(t_t)$ and $T_F(t_t)$ as the outputs, then R_c and t_m are the "levels" of safety that control the process. While we recognize that *test time* is not the only consideration in developing test strategies arrd that there are other important factors, like tbc consequences for reliability and cost, in selecting test cases [6], nevertheless, for the for egoing reasons, *test time* ha\ been found to be strongly positively correlated with reliability growth for the STS [7].

2.3.1 Remaining Failures

We can formulate the risk metric for *criterion 1* as follows:

Risk $R(t_t)$ $(R(t_t)-R_c)/R_c=(R(t_t)/R_c)-1$ **(3)**

Positive, *zero*, and *negative* values of equation (3) correspond to $R(t_t)>R_c$, $R(t_t)$ R_c, and $R(t_t)<R_c$, respectively, which in turn identify the *UNSAFE*, *NEUTRAL*, and *SAFE* regions of operation, respectively.

2.3.2 Time to Next Failure

Similarly, we can formulate the risk metric for *criterion 2* as follows:

Risk Metric $T_F(t_t)$ $(t_m-T_F(t_t))/t_m=1-(T_F(t_t))/t_m$ (4)

Positive, *zero*, and negative values of equation (4) correspond to $T_F(t_t)<t_m$, $T_F(t_t)$ t_m, and $T_F(t_t)>t_m$, respectively, which in turn identify the *UNSAFE*, *NEUTRAL*, and *SAFE* regions of operation, respectively.

Our purpose is to see whether metrics collected during the *static* analysis of the STS flight software (i.e., during requirements inspection) have a quantitative relationship with metrics collected during the *dynamic* analysis of the software (i e, during testing and reliability prediction). We are also investigating how the require.rmaits risk indicators vary with a variety of structural characteristics of the STS code. From these analyses we hope to be able to identify those software-related characteristics of requirements that pose the greatest risk to S"1'S software safely, Th ese analyses, in combination with other methods of assurance, such as inspections, defect prevention, project control boards, process assessment, and fault tracking, provide a quantitative basis for achieving safety and reliability objectives [8].

3. Measuring System Evolution

Our purpose in analyzing software metrics will be to relate measures of software structure to fault content and the dynamic behavior of the system. Specifically, we intend to extend previous work in measuring structural characteristics of software systems to include measuring the effects on fault content and reliability of changes to, additions to, and deletions of code from software [9].

3.1 Establishing a Measurement Baseline

When measuring software evolution, we need to establish a measurement baseline [10]. We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property that, when another point is chosen, the exact same picture of software evolution emerges - only the perspective changes. The individual points involved in measuring software evolution are individual builds of the system.

One problem with using raw measurements is that they are all on different scales. Comparing different modules within a software system by using raw measurement data is complicated by this fact. In order to make such comparisons it is necessary to standardize the data. Standardizing metrics for one particular build is simple. For each metric obtained fm each module, subtract from that metric its mean arid divide by its standard deviation. This puts all of the metrics on the same relative scale, with a mean of zero and a standard deviation of one. This winks fine for comparing modules within one particular build. But when we standardize subsequent builds using the means and standard deviations for those builds a problem arises. The standardization masks the change that has occurred between builds in order to place all the metrics on the same relative scale and to keep from losing the effect of changes between builds, all build data is standardized using the means and standard deviations for the metrics obtained from the baseline system. This preserves trends in the data and lets measurements from different builds be compared.

3.2 Measuring Code Deltas

As a system evolves through a series of builds, its complexity will change. This complexity is measured by a set of software metrics One simple assessment of the size of a software system is the number of lines of code per module. However using only one metric neglects information about the other complexity attributes of the system, such as control flow and temporal complexity. By comparing successive builds on their domain metrics it is possible 10 see how these builds either increase 01 decrease based on particular attribute domains. Using relative complexity, the over all system complexity can be monitored as tbc system evolves [11]. This relative complexity metric has been constructed to serve as a fault surrogate Modules that have large relative complexity values are those most likely to contain softwar C faults

A code delta is, as the name implies, the difference between two builds as computed fos tbc relative complexity metric. These deltas represent how the code has expanded or contracted with respect to the relative complexity metric [12]

The formula for computing code deltas is quite simple. It is given as:

$$\Delta_{metric} = (\rho_B - \rho_A)(X^{AB})^T$$

where PA and ρB are vectors of the relative complexity metrics for tbc modules for tbc previous and subsequent builds of the system and X^{AB} is a vector defined as follows X^{AB} has as many entries in it as their have been modules in the system to this point. That is 10 say, tbc size of X^{AB} is equal to the cardinality of the ad {M1 ∪ M2 ∪ . ∪ MN} where N is the number of builds that have been performed and Mi is the ad of modules involved in build i Each entry in X^{AB} represents a particular software module and is assigned a value of either 1 or O. An entry of 1 indicates that the module was present in either ad A or B and an retry of O is assigned otherwise. For vector subtraction and multiplication to wink, all of the vectors must be of tbc same dimension We will extend the vectors A and B to have the same dimension as the vector X, assigning a 0 to entries corresponding to modules that rue not present in either build When tbc value of A for build 2 is greater than that for build 1 on a particular module, this indicates an increase in relative complexity. Conversely, when the value of build 2 is less than that of build 1 this indicates a decrease in module complexity When a new module appears in build 2 that was not a part of build 1, the value of A frn build 1 is zero and the change is equal to the full value of A for build 2. When a module is present in build 1 but has bcm removed prior to build 2, the value of A for build 2 is zero and the change in tbc system is a decrease by the full value of A for build 1. The sum of these increases and decreases, Δ_{metric}, indicates how the system as a whole has increased or decreased in terms of relative complexity.

3.3 Measuring Code Churn

A limitation of measuring code deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent complexity, the code delta for the system will be close to zero. The over all complexity of the system, bawd on the metric used to compute deltas, will not have changed much However, the reliability of the system could have been sever ely effected by the process of replacing old modules with new ones What we need is a measure to accompany code delta that indicates how much change has occurred Code churn is a measurement, calculated in a similar manner to cork delta, that provides this information

The formula for code churn follows the same method of calculation as code delta with one major exception. Code churn uses the absolute value of the difference in complexity of a module between builds The formula for computing code churn is $V_{metric} = (|B - A|)(X^{AB})^T$ As before, with code delta, when a new module is added in build 2 that was not present in build 1, the value of V for that module in build 1 is zero. However, the converse, where a module in build 1 is not present in build 2, results in the change in complexity being equal to the *positive* full value of the metric for build 1.

When several modules cuc replaced between builds by modules of roughly the same complexity, code delta will be approximately 7L% but code churn will be equal to the sum of the value of V for all of the modules, both inserted and deleted Both the code delta and code churn for a particular metric are needed to assess the evolution of a system These measurements can then be related to tbc rate at which new faults are introduced into the system

3.4 The Profiles of Software Dynamics

To assist in the subsequent discussion of program functionality, it will be useful to make this description somewhat more precise by introducing some notation conveniences Assume that a software system S was designed to implement a specific ad of mutually exclusive functionalities *F*. Thus, if the system is executing a functionality $f \in F$ then it cannot be expressing elements of any other functionality in *F*. Each of these functionalities in *F* was designed to implement a set of software specifications based on a user's requirements From a user's perspective, this software system will implement a specific set of operations, *O*. 'Ibis mapping from the set of user perceived operations, *O*, to a set of specific program functionalities, *F*, is one of the major tasks in the software specification process "l'he software design process is basically a matter of assigning functionalities in *F* to specific program modules $m \in M$, the ad of program modules The design process may be thought of as the process of defining a ad of relations, ASSIGNS over $F \times M$ such that ASSIGNS(f, m) is true if functionality f is expressed in module m.

Each operation that a system may perform for a user may be thought of as having been implemented in a set of functional specifications There may be a one-to-one mapping between the user's notion of an operation and a program functionality. in most eases, however, there may be several discrete functionalities that must be executed to express the user's concept of an operation For each operation, o, that the system may perform, the range of functionalities, f, must be well known Each of the functionalities will exercise a particular subset of program modules

Although we cannot measure time in any meaningful way for execution events of software, we can observe the transition from one module to another. These transition intervals define the notion of the software epoch An epoch begins with the onset of execution in a particular module, and ends when control is passed to another module The measurable event for modeling purposes is this transition among the modules We will count the number Or calls from a module and the number of returns to that module. Each of these transitions to a different module from the one currently executing, will represent an incremental change in the epoch number. Programs executing in their normal mode will make state transitions between modules rather rapidly. In terms of real clock time, many epochs may elapse in a relatively short period

Associat ed with individu al users of a softwar c system is their operational **profile**, or the probability with which they will exercise each of the independent and mutually exclusive operations of the system [13]. This operational profile is a multinomial distribution for each of the operations in tbc set *O*. Wtrm a software system is constructed, it is designed to fulfill a set of functional requirements Users will then run the software to perform a set of operations Typically, tbc operations may not use all of the functionalities with tbc same probability. The **functional** profile of tbc software is the ad of unconditional probabilities of each of the functionalities *F* being executed by tbc user. The probability that the user is executing functionality k is given as $o_k = Pr[Y = k]$, $k = 1,2, ..,\#\{F\}$. A program executing on a serial machine can only be executing one functionality at a time.

When a program is executing a particular functionality, f_k, it will distribute its activity across the ad of modules, M_{f_k} Al any arbitrary epoch, n, the program will be executing a module $m_i \in M_{f_k}$ with a probability u_{ik}. The ad of conditional probabilities u_{*k} where $k = 1,2,..,\#\{F\}$ constitute the execution **profile** for functionality f_k. As a matter of the

program's design, there may be a non-empty set $M_p^{(f)}$ of modules that may or may not be executed when a particular functionality is exercised A particular execution may not invoke any of the modules Of $M_p^{(f)}$, while in other situations all of the modules may participate in the execution of that functionality. This variation in the cardinality of $M_p^{(f)}$ will contribute significantly totbe effort required to test such a functionality.

Most operations will tend to apportion their time across a number of functionalities For a given operation, let / be a proportionality constant $0 \le l_k \le 1$ will represent the proportion of epochs that will he spent executing the k^{th} functionality in $F_k^{(i)}$ Thus an operational profile of a ad of modules will represent a linear combination of the conditional probabilities u_{jk} : $p_i = \sum_{f_k \in F^{(i)}} l_k u_{jk}$.

The manner in which a program will exercise its many modules as the user chooses to execute the functionalities of the program is determined directly by the design of the program Indeed, this mapping of functionality onto program modules is the overall objective of the design process The module profile, *q*, *is* the unconditional probability that a particular module will be executed based on the design of the program It is derived through the application of Bayes' rule First, the joint probability that a given module is executing and the program is exercising a particular functionality is given try $\Pr[X_n = j \cap Y = k] = \Pr[Y = k] \Pr[X_n = j | Y = k] = o_k u_{jk}$ where j and k are defined as before "1'bus, the unconditional probability, q_j of executing module

$\sum_k \Pr[X_n = j \cap Y = k] = \sum_k o_k u_{jk}$ under design j is As was the case for the functional profile and the execution profile, only one module can be executing at any one time Hence, the distribution of q is also multinomial for a system consisting of more *than* two modules

3.5 Functional and Design Risk

Each functionality has an associated execution profile, representing the probability of executing any of the modules under that functionality. As each module is executed, there is an associated cost of execution We will assume that this cost is a function of the module's relative complexity, as relative complexity is a fault surrogate. Each functionality will create an exposure to a potential fauh in a particular module. This exposure is represented by the execution profile. The functional risk ϕ_f associated with each functionality is $\phi_f = \sum_j u_{fj} \rho_j$, which is also the expected value of the relative complexity under the execution profile for functionality *f*. The functional risk nary also be computed for changes to the system between builds in terms of the code delta and code churn measures The functional risk ϕ_Δ for the code deltas is $\phi_\Delta = \sum_j u_{fj} \Delta_j$, while the functional risk c for code churn i. $\phi_\nabla = \sum_j u_{fj} \nabla_j$.

'1 hese functional risk measures arc extremely valuable for regression testing For regression test, we need to identify the functionalities that will maximize the functional risk of either ϕ_Δ or ϕ_∇ This will insure the maximum exposure to the potential faults that were introduced due to the changes in the code between the builds

For a given operational profile, the module profile is determined by the manner in which the system has been designed to distribute its activities among modules Each design alternative will induce a different module profile ϕ_d on the system, where $\phi_d = \sum_j q_i \rho_i$. The Bayes risk for design is one that minimizes operational risk over several design alternatives The reliability of a system running a given functionality is directly proportional to the functional risk If a module has a disproportionate number of faults, then an execution scenario that causes this module to execute a significant number of times will be likely to fail Similarly, tbe overall reliability y of a system may be seen to he directly related to the Bayes risk for the system design

3.6 Characterizing the Development Process

Our experience with software development efforts convinces us that the characteristics of fhe development process exercise quite as great an influence on the number of faults with which the system will be released into operations as do the structural characteristics of the system Therefore, we are characterizing the development process for those systems whose structure we are pleasuring and using statistical methods to relate these measurements to the fault content of the systems being analyzed Several methods of characterizing development processes exist We have elected to usc the characterization scheme developed for the COCOMO **2.0** cost model [4]. There are three reasons for this

1. This information has been shown to be generally useful in constructing software cost models COCOMO81[3] is generally claimed to he accurate to within 30% of the actual cost 70% of the time We might hope to obtain this type of accuracy in predicting fault content
2. The information required for COCOMO 2.0 is generally available for most Exflwaedrwclqn1)011 projects
3. If the data required for COCOMO 2.0 is related to the fauh content of the system, managers will be able to use the same daia to estimate cost and quality, and will make it easier to do cost/quality tradeoffs than is currently possible

We are concerned primarily with the personnel and project attributes used by COCOMO, since we have methods of measuring structure and assessing requirements risk as described show. Personnel attributes describe the skill level and capability of the technical staff responsible for specifying, designing, and implementing the system Project attributes describe the extent to which modern programming practices are used on the project *We* collect this information from participating projects using a questionnaire developed by Boehm at the University of Southern California's Center for Software Engineering Several individuals from each development effort are interviewed during the course of completing the questionnaires to assure that the responses arc as complete and unbiased as possible.

4. integrating Risk and Metrics Analysis

We are investigating how to integrate the methods of assessing risk and measuring the evolution of a software system as described above into a predictor of the fault content of the system at any point in the future. This predictor should address the following general limitations of current software reliability models

1. Currently available techniques do not simultaneously take a system's structure, its development process characteristics, risk factors, and calendar time into account Although some methods exist to predict error density during the early phases of a development effort [14, 15, 16, 17], they make very restrictive assumptions about development process characteristics, relate only structural characteristics of the system to fault insertion, predict what the fault content will be at the end of a development phase rather than at an arbitrary time, a predict what the error density will be at the start of system testing
2. With the exception of the framework developed by Rome Laboratories [14], none of the currently-available predictors for the early development phases relates static measures to reliability y or a measure directly related to reliability (c g., Mean Time To Failure, Hazard Rate). Although the relationship between error density and failure intensity developed for the Rome Laboratories framework can be used, it nray produce inaccurate reliability predictions. This is because the reliability of a software system depends on the system's operational profile [18, 19]. For a software system, the input to that system determines whether or not a fault will be exposed, and whether that fanh exposure will result in a faihue If the system's operational profile is such that the most frequent inputs are those that will expose faults, the system will be appear to be unreliable However, if the operational profile is such that most of the inputs to a system do not expose faults, the system will appear to be reliable Work in this field indicates that determination of the system's dynamic characteristics cannot be accomplished by simple analysis of the system's requirements and design [19], In view of this situation, it may not be possible to develop a predictor that produces reliability or reliability-related estimates from information available prior to the test phases However, since static measures such as fault counts do not depend on the environment in which the software executes, it should he possible to develop a predictor that will make predictions of this type If information about the

dynamic characteristics of the system is available, we may then be able to extend the static predictor to make reliability estimates and forecasts.

3. The predictor should be sufficiently adaptable to use the measurements of software structure, development process characteristics, and risk that are available during each phase in a development effort. For instance, development process characteristics that are available early in a development effort would include estimates of the development schedules, staffing profile estimates, and a plan for phasing the development effort. As far as structural characteristics are concerned, function points or object points might be considered for the early phases of a development effort, as might the complexity categorizations associated with the COCOMO 2.0 software cost model, while more detailed structural information would be available in later phases. Many of the types of assessments of risk described in Section 2 are available during the requirements and architectural design phases (e.g., qualitative assessment of the complexity of a change, number of requirements affected by a change, size and number of data structures affected by a change).

We propose a predictor in the form of a series of linked birth and death models that can be used to estimate total fault content. The expressions for the rates would be functions of the system's structural characteristics, its development process characteristics, risk assessment, and the number of faults already present in the system. The predictor would use a birth and death model for each life cycle development phase to make estimates of the expected number of errors in a work product produced during that phase. In addition to the expected number of errors, the probability of the product containing a particular number of errors would also be available. This probability distribution, together with information about how the remaining errors multiply during the next phase, would be used as input to the birth and death model representing the next phase of development.

5. Conclusion and Future Work

Many of the risk indicators that we are relating to reliability and fault content are quantitative - for instance, the number of lines of code affected by the change, the effort required to implement the change, and the size of data and code areas affected by the change can all be measured or estimated. Other risk indicators, such as the perceived complexity of a change to an existing system, can at least be assigned levels according to well-established categorization methods [3, 4]. However, there are some important risk factors that are entirely qualitative. For example, conflicts between requirements changes are an important source of faults. We are not aware of any method by which the type and extent of these conflicts could be meaningfully quantified within the scope of our current effort.

In the area of metrics analysis, our current work involves measuring the evolution of various NASA software systems to obtain measures of code delta and code churn. These systems include selected portions of the S1'S flight software, on-board control software for unmanned spacecraft currently under development, and the ground systems for several unmanned spacecraft. Using information from the problem tracking systems used during test, we can estimate the point at which a fault was first inserted into the code. By doing this, we obtain both estimates of the rate of fault introduction, and validate the use of the code delta and code churn measures as surrogates for fault count. Having obtained values for these rates, we can then relate them to the development process characteristics and measures of risk. Although we are doing this only for the implementation phase, we do not see any reason why we could not extend this type of analysis to the earlier development phases. However, we have intentionally restricted ourselves to analyzing the evolution of the source code of each software system. This is for the following reasons:

1. While there is a wide variety of automated tools to measure source code, there are few tools available for measuring requirements and design documentation.
2. Requirements and design documentation, at least for the efforts we are studying, tend to be in forms that cannot be easily measured. Requirements especially tend to be in natural language or semi-formal notation that cannot easily be measured without expending great effort in translating the requirements into a formal notation that can be read by automated tools.
3. Unlike source code, requirements and design documentation is not usually managed by revision control systems such as SCCS and RCS. This makes it difficult to identify any particular "build" of the requirements or design, making any measurements analogous to code delta and code churn during these phases extremely difficult as well as rendering them suspect.

At this point, we examine how well the predictor described in Section 4 addresses the limitations on predictive models given at the beginning of this section.

1. Currently available techniques do not simultaneously take a system's structure, its development process characteristics, risk factors, and calendar time into account. The predictor described above takes all of these factors into account. In particular, it takes elapsed time into account, allowing predictions to be continuously updated throughout the development effort. If the development process, product characteristics, or indicators of risk change during a development effort, predictions can be easily updated. For example, suppose that the development characteristics remain constant from project start to time τ. The development process then changes at time τ, and remains the same until the start of testing. This model would allow predictions of the remaining number of errors at time τ to be made. Predicting of the number of faults at the start of testing would be done as follows:
   i. Use the model to predict the number of faults remaining at time τ. This is simply $\mathbf{P}(\tau) * \mathbf{I}$, where $\mathbf{I}$ is the initial state vector specifying the discrete probability density function for the number of faults in the system. This would be a vector $\mathbf{u}$ of length $|\mathbf{P}(\tau)|$ - the xth entry of $\mathbf{u}$ would denote the probability of x faults remaining in the system.
   ii. Using the new development process characteristics, predict how many faults will be added between τ and the start of test. Since the model explicitly deals with rates, this is quite straightforward. First, select τ as the starting point of the predictions. Next, write down the new rate matrix $\mathbf{A}$. Compute the probability transition matrix $\mathbf{P}(t-\tau)$ as $e^{\mathbf{A}(t-\tau)}$, where t ranges from τ to the start of system test. The number of faults remaining at the start of test is simply $\mathbf{P}(t-\tau)*\mathbf{u}$, where $\mathbf{u}$ is the predicted number of faults remaining at time τ, described above.
2. With the exception of the framework developed by Rome Laboratories [14], none of the currently-available predictors for the early development phases relates static measures to reliability or a measure directly related to reliability (e.g., Mean Time To Failure, Hazard Rate). This is still a limitation for this predictor. The predictor described in this section returns predictions related to fault counts rather than reliability. The limitation exists because the rate expressions do not use any information about the dynamic behavior of the system. Even if this information were available, including it in a form that could be used in this type of predictor would be a significant challenge. Our experience indicates that collection of the information is a significant task, and is appreciably more involved than the collection of the information required to calibrate the predictor described above.
3. The predictor should be sufficiently adaptable to use the measurements of software structure, development process characteristics, and risk that are available during each phase in a development effort. Since the predictor is in the form of a series of linked birth and death models, one such model for each development phase, each stage of the series uses the measurements that are available for a specific development phase. As in point (1) above, we see that predictions can easily be updated as newer or more accurate information becomes available.

We intend to address the following areas in future work:

1. Find ways to better quantify risk indicators. One possibility might involve using formal methods to specify selected portions of the system under study. It may be possible to combine formal methods with FTA on new requirements and new software. If formal methods were to be used on a production basis or on a selected basis on new software, there would be a consistent type of requirements specification language and format to which FTA could be applied.
2. Extend the work in risk analysis to include relating risks identified at the system level to software reliability and fault content. As above, we might examine the use of FTA and formal methods to identify hazards at the system level that could be related to risks in the software component.
3. Extend the ability to measure the evolution of a software system to the design and requirements phases. This would require finding ways to track the evolution of a software system as it is being specified, designed, and implemented to deal with the current problem of not being able to identify 'builds" well enough to measure quantities analogous to code delta and code churn. Development organizations that are sufficiently mature might retain historical records of technical

reviews from which the evolution of the requirements and design might be reconstructed Also, ways of documenting the requirements and design would have to be found that would make it possible to produce objective measures of system structure. One possibility would be to rewrite the specifications using formal methods The structure of the specification could be measured with enhanced versions of tools that were developed as part of previous work at the Jet Propulsion Laboratory [20].

Acknowledgment

The work described in this paper was carried out at the Jet Propulsion Laboratory, the Naval Postgraduate School, and the University of Idaho through funding from the NASA IV&V Facility in West Virginia

References

[1] Norman F. Schneidewind, 'Reliability Modeling for Safety Critical Software", *IEEE Transactions on Reliability*, December 199&

[2] Norman F. Schneidewind, "Methodology for Validating Software Metrics", *IEEE Transactions on Software Engineering*, May, 1992.

[3] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[4] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, R Selby, "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," *Annals of Software Engineering*, volume 1, J.C. Baltzer Science Publishers, Amsterdam, The Netherlands, 1995, pp. 57-94.

[5] N. G. Leveson, "Software Safety: What, Why, and How", *ACM Computing Surveys*, Vol 18, No. 2, June 1986, pp. 125-163.

[6] Elaine J. Weyuker, "Using the Consequences of Failures for Testing and Reliability Assessment", *proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, D.C., October 10-13, 1995, pp. 81-91.

[7] Norman F. Schneidewind and T. W. Keller, "Application of Reliability Models to the Space Shuttle', *IEEE Software*, Vol 9, No. 4, July 1992 pp. 28-33.

[8] T. Keller, N. F. Schneidewind, and P. A Thornton, "Predictions for Increasing Confidence in the Reliability of the Space Shuttle Flight Software", *proceedings of the AIAA Computing* in *Aerospace* 1@, San Antonio, TX, March 28, 1995, pp. 1 -S.

[9] J. C. Munson and "P. M Khoshgoftaar "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, S1,~18, No. .§1992, pp. 423-433,

[10] J. C. Munson and D. S. Werries, "Measuring Software Evolution," *Proceedings of the 1996 IEEE International Software Metrics Symposium*, IEEE Computer Society Press, pp. 41.51,

[11] J. C Munson, "Software Measurement: Problems and Practice," *Annals of Software Engineering*, Vol 1, No. 2, J.C. Baltzer AG, Amsterdam 1995, pp. 255-285.

[12] J. C. Munson and G. A Hall, "Dynamic Program Complexity and Software Testing," *Proceedings of the 1995 IEEE International Test Conference*, IEEE Computer Society Press, pp. 730-737,

[13] J. C. Munson, " A Functional Approach to Software Reliability Modeling," will appear *Proceedings of the Quality of Numerical Software Assessment Conference*, Chapman and Hall

[14] J. McCall, J. Cavano, "Methodology for Software Reliability y Prediction and Assessment," Rome Air Development Center (RADC) Technical Report RADC-TR-87-171. vol 1 and 2, 1987.

[15] J. E. Gaffney, Jr. and C. F. Davis, "An Approach to Estimating Software Errors and Availability," SPC-TR-88-007, version 1.0, March, 1988, *proceedings of Eleventh Minnowbrook Works hoi) on Software Reliability*, July 26-29, 1988, Blue Mountain Lake, NY,

[16] J. E Gaffney, Jr. and J. Pietrolewicz, "An Automated Model for Software Early Error Prediction (SWEEP)," *proceedings of Thirteenth Minnowbrook Workshop on Software Reliability*, July 24. 27, 1990, Blue Mountain Lake, NY,

[17] J. C. Kelly, J. S. Sherif, J. Hops, "An Analysis of Defect Densities Found During Software in.ywclimts" , *Journal of Systems Software*, vol 17, pp 111-117, 1992,

[18] John D. Musa , Anthony Iannino, Kazuhiro Okumoto, Software Reliability: Measurement, Prediction, Application; McGraw-Hill, 1987; ISBN O-07-044093-X,

[19] M Lyu, ed , Handbook of Software Reliability Engineering, McGraw-Hill, 1996; ISBN.0-07-039400-8.

[20] A P. Nikora, R G, Covington, J. C. Kelly, W. J. Cullyer, "Measuring the Complexity of Formal Specifications", funded by the JPL Director's Discretionary Fund, 1992-1993.