ON BAYESIAN ESTIMATION FOR SOFTWARE RELIABILITY

Yang Y
US NRC
11555 Rockville Pike, Rockville, MD 20852, USA
Yaguang.yanag@nrc.gov

ABSTRACT

System safety is closely related to system reliability. Safety requirements many times are translated to reliability requirements. Nowadays, software systems exist in many engineering systems. However, there is no consensus method for software reliability estimation. On the other hand, there is an increasing interest in estimating the software reliability due to concerns for safety critical systems. In this paper, we try to close the gap by proposing a systematic and probabilistic method to estimate the software reliability based on software test data.

Key Words: Bayesian method, reliability, software, flow network

1 INTRODUCTION

System safety for safety critical systems is now a part of engineering design [13]. For example, application of probabilistic safety assessment (PSA) is required for nuclear power plants designs [11][12]. However, gaps exist between these design requirements and lacking of applicable methods and tools to implement the requirements. It is noticed in [12] that “at present it is not possible to derive a probabilistic model of software failure”. In reality, safety critical systems rely heavily on the equipment with digital instrumentation and control (DI&C) components/systems which include software. Therefore, there is increasing interest in international organizations, such as the United States Nuclear Regulatory Commission, in reliability and safety analyses for safety-critical DI&C systems, in particular, in the reliability analysis for software systems.

Many attempts have been made for more than four decades to find practical ways to improve software reliability and to estimate the software reliability. Lawrence [17] discussed activities that should be carried out throughout the software life cycle. Parnas, Asmis, and Madey [26] emphasized documentation requirements and quality control, including testing and reviews. Leveson and Harvey [18] proposed software fault tree analysis method. Up to the date, the most extensively investigated software reliability method is the software reliability growth model [6]. Many statistical models, such as the exponential distribution, Poisson distribution, Weibell distribution, and Gamma distribution models have been considered, and many different scenarios have been discussed in [10,14,15,26,30] and references therein. The reliability growth model is more like traditional hardware model in that the failure is time-related. However, unlike hardware-only analog equipment which is due to aging and weariness, therefore, failures are time-related, the software failure mechanism is due to undetected human errors, therefore, failures are not time-related. For this reason, the reliability estimation method for a software system should be different from that used for hardware.

* Office of Research, US NRC.
Because typical software failures are due to undetected human errors in certain parts of the software and failures are triggered by combinations of specific events and input data sets. A more attracted and reasonable estimation for software failure should be test-based probabilistic method, which was first introduced in [23], where a black-box\(^1\) model was proposed. This idea was further studied by many researchers in more recent efforts [3,4,19,20,21,31] for different scenario. These methods consider typically black box situation. There are some drawbacks for using the black-box model because they do not consider the complexity of the software systems. For example, if two different codes have the same number of successful test, black-box method gives the same estimated reliability value even if the sizes of the two codes are completely different or if the sizes are the same but the complexity of the structure of the two codes are drastically different.

Noticing the drawback of the block-box methods, some researchers considered the architecture-based software reliability analysis (see [8,9] and numerous references therein). Although software architectures are considered in this type of methods, these methods consider the architecture in software component level, the structure is at high level that does not provide the detailed structure information inside the software program. Beside, components level reliability estimation is not rigorously based on the probability theory.

For the aforementioned reasons, there is no consensus on an ideal software reliability estimation method [12]. Because of this difficulty, some nuclear plant probability risk analysis (PRA) assumed that software reliability is some constant without giving a method to validate the number, for example, ESBWR (economic simplified boiling water reactor) PRA analysis [1], which is far less than ideal.

In this paper, we restrict our discussion to single thread software, we also assume that the safety critical software under the discussion follows the guidelines\(^2\) in Section 4.4 of [18]. These assumptions on one hand make software structure simple, logic clean, and easy to analyze; on the hand, these assumptions should cover a class of software that are used in safety critical systems, for which design simplicity, deterministic execution time, and high reliability are required or desired. We propose some refinements for a recently developed software reliability estimation method given in [32,33] which is based on test data and software structure. The method is so promising that a hierarchical structure modeling platform has been built for nuclear system analysis [2]. The refinement in this paper makes method rigorously based on probability theory and test data similar to the methods in [19,20,21,23,31] while making full use of the structure of the software as we did in [32,33].

The remainder of the paper is organized as follows. Section 2 presents a flow network model for software structure and a simple example is given to describe it. Section 3 introduces a software reliability estimation method with a Bayesian estimation refinement. Section 4 discusses an automated tool for software reliability calculation. The last section gives the conclusions of this paper.

\(^1\) By black-box, we means that the software is viewed as a whole piece of code without considering the details inside the software.

\(^2\) Many guidelines for safety critical software are provided in [17], such as not using assembly language, not using recursion, interrupt, exception, pointer or indirect addressing, unconditional branches, branching out of loops, etc.
2 SOFTWARE COMPONENT FAILURE MODEL

There is no consensus on a method to be used for software reliability assessments due to lack of objective and rigorous methods, because most existing software reliability estimation methods use models suitable for hardware failures, while software failures are fundamentally different from hardware failures. Typical hardware failures are due to wear out (an example is the moving parts in mechanical systems) or aging-related (an example is mechanical parts made by plastic materials) failures. Therefore, hardware failure models are based on a random failure time that can be described by exponential, Poisson, Weibull, or Gamma distributions. However, typical software failures are due to undetected human errors in certain parts of the software and failures are triggered by combinations of specific events and input data sets. A problematic line of code in a software path may or may not be executed because the path is executed only when certain conditions are met. We call these conditions that determine the execution of a problematic path are triggering events. Even a problematic line of code is executed, the software may or may not fail because the data set used in the problematic line of code may not cause problem, for example, “c=a/b” does not cause any problem if b is not zero. Therefore, a failure occurs when triggering events determine the software to execute a problematic part of the software and a triggering data set is in use. Almost all existing methods that address software reliability issues do not model software failure in an appropriate way. Most software reliability models, such as the most extensively studied reliability growth model, are based on a random failure time which is inappropriate. Our model will consider both triggering events and the probability of existence of human mistakes.

2.1 Software Structure Model

There are several reasons that the software structure should be considered in the software failure model. First, the software structure is related to the triggering events which decide the path of software execution. Second, the complexity and details for every individual software is represented in the flow network model. Third, testing information can be fully used in the reliability analysis. Finally, a complex software can be divided into many simple pieces of codes; there is no branches inside each piece of the code, therefore, estimation reliability for each piece of code is easy to handle mathematically.

We use a flow network (also named as control flow graph (CFG) in software engineering literature [22]) to model the software structure. Let source denote the start point of the software; sink denote the end of the software; n nodes \( n_i \in N \) represent the logic branch or converging points; and edges \( e_{ij} \in E \), \( j = 1, \ldots, m \) denote the software code between node \( i \) and the node next to \( i \). If an edge is executed, then every line inside the edge is executed; i.e., no branch exists inside an edge. It is assumed that there is an infinite capacity in every edge, which means that each edge can have as many tests as desired. Using c/c++ language as an example, the nodes are collections of the beginning and the end of every function, the beginning and the end of every conditional block starting with ‘if’ or ‘switch’; while the edges are collections of pieces of software between nodes that meet one of the following conditions: (a) between the lines of the start of each function and the first ‘if’, ‘switch’ or ‘while’; (b) between the lines ‘if’ and the line ‘else’ or ‘else if’ or the end of ‘if’, or between the line ‘else if’ and the next ‘else if’ or the line that ends ‘if’; (c) between the lines of ‘case’; (d) between the lines after the end of ‘if’ or the line after the end of ‘switch’ and the line before the next ‘if’ or ‘switch’ or the line that ends the function. We use the following simple pseudo c/c++ code to describe the partition and the flow network concept.
Main()
{
    Data initialization;
    If condition A holds
    {
        Process data;
        If data process success
        {
            Save result;
        }
        Else if data process fail
        {
            Issue a warning;
        }
    }
    Else if condition A does not hold
    {
        Print "condition fail";
    }
    Clean memories;
}
It is worthwhile to point out that (a) any node is part of the edge right before the node, (b) there are software toolset capable to convert the software into the flow network [16], therefore, the translation from software to flow network will not be a technical obstacle to use the proposed model even for large software systems.

It is clear that the software structure represented by the flow network is related to the triggering events (checked at nodes) which decide the path of software execution. Also, the complexity [22] and details for every individual software is represented in the flow network model. The complexity and details will be fully considered in the next section when we discuss how to estimate the reliability of the entire software system.

To make the flow network model useful in the software reliability estimation, we need to perform test and record the numbers of successes and failures so that the statistical data can be used to estimate the reliability. After each test, the software development environment will record the software execution path of the test, say for a test the path is $e_{11} \rightarrow e_{21} \rightarrow e_{32} \rightarrow e_{41}$, the software engineer will examine the test result. If the test result meet the expectation, the engineer enters a key to accept the success of the test, the success numbers for all the edges on the path, namely, $e_{11}, e_{21}, e_{32},$ and $e_{41}$, will be increased by one. In a different test with different test data set, if the software execution path in the test is the same, but the result does not meet the expectation, the software engineer will identify the defect, say in edge $e_{32}$, then the engineer enters a key to reject the test result, the software development environment will set the success number of $e_{32}$ to zero because this edge was modified while keep the success numbers of other edges the same because they are not touched. Therefore, useful test information for the edges $e_{11}, e_{21},$ and $e_{41}$ are not lost. This information will be used in the software reliability estimation discussed in the next section. It is worthwhile to point out there are tools that are designed to record the test path [29].

Because each piece of code (edge) is very simple, there is no branch inside, it is easy to model the probability of existence of defect in an edge, which is our next topic.

### 2.2 Software Defect Model

Note that, for any piece of code, there are only two possibilities: either it has defect(s) or it has no defect. Therefore, we can use Bernoulli distribution to model if a piece of code $e_y$ has defect(s) or not, i.e., a binary random variable $D_y$ is used to represent the probability of the existence of defect(s) in a piece of code. Let $D_y = 1$ denote an event in which edge $e_y$ has been executed once and where the test result meets the expectations, therefore, $p_y = P(D_y = 1)$ denote the probability of no defect in this piece of software (there is a probability that a defect exists but is not detected). Let $D_y = 0$ denote an event in which edge $e_y$ has been executed once and where the result fails to meet the expectations because of some defect(s), therefore, $q_y = P(D_y = 0) = 1 - p_y$ is the probability of existence of at least one defect in this piece of code (there is a probability that no defect exists but the test engineer thinks that s/he finds a defect). Therefore, $p_y$ is the probability
of having no defect in \( e_y \) and \( q_y \) is the probability of having at least one defect in \( e_y \). Clearly, the one-time test scenario follows a Bernoulli distribution. It is likely that some edges are executed more than once in the software test stage. These multiple-test scenarios follow a binomial distribution. The main idea of this paper is to determine the probability \( p_{ij} \) by using repeated software test and examine the test results to see if a piece of code passes a test or not. If a piece of software contains some defect(s), then there is a probability that the software with defect(s) will fail if some triggering event occurs and if a triggering data set is in use. Moreover, examining the test result may catch the defect (when this occurs, the defect in the software will be fixed and the software quality will be improved). When testing is finished, the reliability of a piece of software is estimated according to the number of times that this piece of software passes the test after the last modification.

Since a piece of code may be tested many times involving different data set, we model the existence of defect(s) in a piece of software \( e_y \) by using a binomial distribution. Given a piece of code \( e_y \) which has \( h_{ij} \) executions in the test stage, then probability that this piece of code passes test \( k \) times follows a binomial distribution,

\[
p(E_{ij} = k) = C_{h_y}^k p_y^k q_y^{h_y-k},
\]

where \( k = 0,1,2,\ldots, h_y \), and

\[
C_{h_y}^k = \frac{h_y^!}{k!(h_y-k)!}.
\]

In the remainder of the paper, we propose and derive a test-based Bayesian estimation method to estimate reliability of each piece of the code, which is one of the major differences from the method discussed in [32][33]. We also take the structure of the software into consideration in software reliability estimation because (a) it reflects the individual software complexity, and (b) tests are not equally executed in every line of code. This lower level of detail should give better reliability estimations than the black box model [5] because more information is used in the estimation. To simplify our presentation and save space, we focus on software which is used in safety critical systems and follows the guidelines in [18].

### 3 SOFTWARE RELIABILITY ESTIMATION

There are two major differences in software reliability between the method proposed in this paper and the method proposed in [32,33]. First, the reliability estimation for a single edge proposed here uses rigorous Bayesian method while the method discussed in [32,33] uses a subjective heuristic method. Second, the method proposed here estimates the software reliability directly and the method discussed in [32,33] estimates failure rate and then translates to the reliability.

#### 3.1 Reliability Estimation for a Single Edge Using Bayesian Method

Let \( T_{ij} \) be the average execution time of the edge \( e_y \) of the software and \( h_{ij} \) be the total number of executions of \( e_y \) in the software test stage. If an edge \( e_y \) of the software is executed in a test
scenario, we consider that the test scenario covers the edge $e_{ij}$. As we discussed above, some edges may be tested more times than other edges. A main question in a multiple-test scenario is how to determine probability of no defect or reliability in these situations. The proposed reliability estimation is based on the test result using the Bayesian method. Let $t_i$ be the outcome of the $i$th test. The posterior density of $p_{ij}$ is sourced from earlier work, as follows [28]:

$$p(p_{ij} \mid t_1, \ldots, t_{h_{ij}}) = \frac{p(t_1, \ldots, t_{h_{ij}} \mid p_{ij})g(p_{ij})}{f(t_1, \ldots, t_{h_{ij}})} = \frac{p(t_1, \ldots, t_{h_{ij}} \mid p_{ij})g(p_{ij})}{f(t_1, \ldots, t_{h_{ij}})} , \quad (3)$$

where

$$f(t_1, \ldots, t_{h_{ij}}) = \int p(t_i \mid p_{ij}) \Lambda p(t_{h_{ij}} \mid p_{ij})g(p_{ij})dp_{ij} , \quad (4)$$

and $g(p_{ij})$ is a conjugate prior of distribution, which, for binomial distribution, is a Beta distribution with a pair of parameters $(\alpha_{ij}, \beta_{ij})$ [7,25] where $\alpha_{ij} > 0$ and $\beta_{ij} > 0$ are chosen to reflect any existing belief or information about $e_{ij}$. If the piece of code $e_{ij}$ has been tested $n$ times, among these tests, $e_{ij}$ has passed the test $k$ times, then, the posterior density is also a Beta distribution with the parameter $(\alpha_{ij} + k, \beta_{ij} + n - k) > 0$ which is given as

$$p(p_{ij}) = \frac{1}{B(\alpha_{ij} + k, \beta_{ij} + n - k)} p_{ij}^{k+\alpha_{ij}-1}(1-p_{ij})^{n-k+\beta_{ij}-1} , \quad (5)$$

where $B(\alpha_{ij} + k, \beta_{ij} + n - k)$ is the Beta function. Given $n$ tests with $k$ success incident, the reliability estimation for a piece of code $e_{ij}$ is the mean of $p_{ij}$ which is given by

$$R_{ij} = E(p_{ij} \mid k, n, \alpha_{ij}, \beta_{ij}) = \frac{\alpha_{ij} + k}{\alpha_{ij} + \beta_{ij} + n} . \quad (6)$$

In software testing practice, if a test does not meet the expectation, an investigation is conducted to find the defect and a fix is made. The total test and successful times is reset$^3$. Therefore, we always have $n = k = h_{ij}$. But we may adjust parameters $(\alpha_{ij}, \beta_{ij}) > 0$ after a fix is made since we gain confidence as the software matures. We can do so by increase $\alpha_{ij}$ or decreasing $\beta_{ij}$. To calculate failure probability, we simply use

$$q_{ij} = 1 - p_{ij} . \quad (7)$$

Therefore, at the end of the test stage, the software reliability can be modeled by a flow network whose structure is represented by nodes and edges, moreover, each edge has its reliability (probability of no defect) determined by the test results and is estimated by (6). Based on this observation, it becomes possible to simplify the software reliability model by repeatedly combining either serial-connected edges or parallel-connected edges into a combined artificial edge until we reduce the entire software to a single edge whose reliability is then the software reliability. In the following discussions, we present the procedures and details pertaining to the combining of parallel-connected edges or serial-connected edges into a single artificial edge; we will also provide formulae to estimate the reliability of the combined artificial edge depending on whether these edges are serial-connected or parallel-connected edges.

$^3$ Except the retest for the failed cases, each test should be created independently.
3.2 Reliability Estimation for Parallel Edge

Unlike serial edges, parallel edges exist naturally in software because of statements such as “if-else” or “case” etc. During the test stage, each edge under the same node may not be executed the same number of times and the execution time for each edge can be different. Therefore, when we combine the parallel edges into a single edge, we should weight the probability of each edge differently according to the number of execution times and average execution times (because these considerations are related to operational profile [24]). For an edge that execute more times than other edges in the test stage, we should give it a heavier weight than other edges. Similarly, for an edge that need more time to finish the execution, we should give it a heavier weight than other edges. First, for a block under node $n_i$ composed of $m_j$ parallel-connected edges, the total number of executions of all parallel-connected edges $e_{ij}$ during the test stage is

$$h_i = \sum_{j=1}^{m_j} h_{ij}$$ (8)

and the total execution time of all parallel-connected edges $e_{ij}$ under node $n_i$ during the test stage is the summation of the execution time multiplied by the number of executions of every edge, i.e.,

$$T_i = \sum_{j=1}^{m_j} T_{ij} h_{ij}.$$ (9)

Given that edge $e_{ij}$ has $h_{ij}$ executions in the test stage and each parallel edge with multiple tests follows a binomial distribution,

$$(p_{ij} + q_{ij})^{h_{ij}} = \sum_{k=0}^{h_{ij}} C_{h_{ij}}^k p_{ij}^k q_{ij}^{h_{ij}-k} = 1$$ (10)

holds for every parallel-connected edge immediately under node $n_i$. As every edge in the parallel structure has its own binomial distribution $(p_{ij} + q_{ij})^{h_{ij}}$ and its own execution time $h_{ij} T_{ij}$, the binomial distribution for the entire parallel structure which has a total execution time of $T_i$ should be a convex combination of the binomial distributions of individual edges weighted by $\frac{h_{ij} T_{ij}}{T_i}$; i.e., an edge that has longer execution time and executed more times has a larger weight, and the summation of all of the weights is $\sum_{j=1}^{m_j} \frac{h_{ij} T_{ij}}{T_i} = 1$. Therefore, the distribution of parallel edges should satisfy, considering (9) and (10), the following relationship:

$$\sum_{j=1}^{m_j} \frac{h_{ij} T_{ij}}{T_i} (p_{ij} + q_{ij})^{h_{ij}} = \sum_{j=1}^{m_j} \frac{h_{ij} T_{ij}}{T_i} = 1.$$ (11)

Hence, immediately following (11), the reliability of the block composed of the parallel-connected edges under node $n_i$ (i.e., the probability of no defect for the edges under the node $n_i$) is the event that every test has successfully passed. The probability of this event is given by

$$R_{i1} = \sum_{j=1}^{m_j} \frac{h_{ij} T_{ij}}{T_i} R_{y_j}$$ (12)
The number of executions of the combined artificial edge used in the next model reduction step is taken as the summation of the number of the executions of all of the edges in the parallel block,

$$H_{ji} = \sum_{j=1}^{i} h_{ij} \quad (13)$$

The equivalent execution time for the combined artificial edge (from the parallel block) used in the next model reduction step is

$$T_{ji} = \frac{1}{H_{ji}} \sum_{j=1}^{i} T_{ij} h_{ij} \quad (14)$$

The same method can be applied to the parallel-connected blocks, including blocks that are reduced to artificial edges.

3.3 Reliability Estimation for Serial Edge

Serial edge connection does not exist in the initial flow network or CFG. However, if a single edge is connected to a set parallel edges, after we merge the parallel edge into a single edge, it creates a connection of two serial edges. Therefore, we need to have a formula to combine the serial connected edges and be able to estimate the reliability of the combined edges. For a block under node $n_i$, composed of nodes $i_1, K, i_s$ and serial-connected edges (assume that there are no parallel-connected edges in all nodes $i_1, K, i_s$), the total number of executions of all serial-connected edges $e_{ij}$ during the test stage is $h_i = h_{ij}$ for any $i \in i_1, K, i_s$, and the total execution time of all serial-connected edges $e_{ij}$ under node $n_i$ during the test stage is the summation of the execution time multiplied by the number of executions of every edge, i.e.,

$$T_i = \sum_{i=1}^{i_1} T_{ij} h_{ij} \quad (15)$$

Given that edge $e_{ij}$ has exactly $h_{ij}$ executions in the test stage and each serial edge with multiple tests follows a binomial distribution,

$$(p_{ij} + q_{ij})^{h_{ij}} = \sum_{k=0}^{h_{ij}} C_{h_{ij}}^k p_{ij}^k q_{ij}^{h_{ij}-k} = 1 \quad (16)$$

holds for all serial-connected edges immediately under node $n_i$. For the serial-connected edges, the reliability of the entire block is the minimum of the reliabilities of the individual edges. Considering (14), the following relationship immediately holds:

---

4 For serial-connected edges, the method of [32,33] uses the product of the reliabilities of the individual edges as the reliability of the serial-connected edges. A drawback of using the product of the reliabilities of the individual edges is the inconsistence in the following scenario: if we artificially divide an edge into a set of serial-connected edges, then the reliability for the edge and the reliability for the set of serial-connected edges using formula involving the product of the reliabilities of the individual artificial edges are not consistent. But using the minimum for this scenario will lead to a consistent presentation. Another reason that a minimum is used in (18) is because when a test is successful and one of serially connected edge is in the path, all serially connected edges are in the path and all these edges pass the test.
\[
\prod_{i \in n_k} (p_{ij} + q_{ij})^{h_{ij}} = 1. \tag{17}
\]

Hence, the reliability of the block composed of serial-connected edges under the node \( n_k \) in the software (i.e., the probability of no defect for the block composed of serial-connected edges under the node \( n_k \)) is the event that every test has successfully passed. The probability of this event is given by

\[
R_{n_k} = \min(R_{ij}). \tag{18}
\]

The number of executions of the combined artificial edge used in the next model reduction step is taken as the number of executions of any edge \( h_{ij} \), where \( i \in i_s, K, i_s \), in the serial block, and

\[
H_{h_{ij}} = h_{ij}. \tag{19}
\]

The equivalent execution time for the combined artificial edge (from the serial block) used in the next model reduction step is

\[
T_{h_{ij}} = \sum_{i \in n_k} T_{ij}. \tag{20}
\]

The same method can be applied to the serial-connected blocks, including blocks that are reduced to artificial edges.

### 3.4 Overall Reliability Estimation for the Software

The overall reliability of the software is estimated as follows. First, construct the flow network as discussed in Section 2. As testing proceeds, repeatedly use (6) to update the reliability for each edge. The reliability of each edge is obtained when the test finishes. Given the reliabilities of all of the edges, one can use equations (12-14) to simplify parallel-connected edges into a single artificial edge and use equations (18-20) to simplify serial-connected edges into a single artificial edge. The software reliability is obtained by repeating the process until all of the edges are combined into a single artificial edge. We then obtain the total equivalent test time \( T \) and the software reliability \( R \). An example is used to describe the process in the next subsection.

### 3.5 An Example

The pseudo c/c++ code example introduced in Section 2 is used to demonstrate how this software reliability estimation method works. The software partitioned as in Fig. 2 (a) has five nodes and six edges. Assume also that three tests are conducted. The first test path is \( e_1e_2, e_3, e_4, \) the second test path is \( e_1e_2, e_3e_4, \) and the third test path is \( e_1e_2e_4. \) Assume further that the total test time is \( T = .00011 \) hours and \( T_y = .00001 \) hours for every edge. Therefore, \( h_{11} = h_{41} = 3, \ h_{21} = 2, \) and \( h_{22} = h_{31} = h_{32} = 1. \) Assume

\[
R_{11} = \frac{\alpha_{11} + h_{11}}{\alpha_{11} + \beta_{11} + h_{11}} = \frac{\alpha_{41} + h_{41}}{\alpha_{41} + \beta_{41} + h_{41}} = R_{41} = 0.9999999 \quad \text{for} \quad e_1 \text{ and } e_4; \]

Similarly assuming

\[
R_{31} = R_{32} = R_{22} = \frac{\alpha_{31} + h_{31}}{\alpha_{31} + \beta_{31} + h_{31}} = \frac{\alpha_{32} + h_{32}}{\alpha_{32} + \beta_{32} + h_{32}} = \frac{\alpha_{22} + h_{22}}{\alpha_{22} + \beta_{22} + h_{22}} = 0.99 \quad \text{yields, and assuming}
\]

\[
R_{21} = \frac{\alpha_{21}}{\alpha_{21} + \beta_{21}} = R_{22} = \frac{\alpha_{22}}{\alpha_{22} + \beta_{22}} = 0.99.
\]
\[ R_{21} = \frac{\alpha_{21} + h_{21}}{\alpha_{21} + \beta_{21} + h_{21}} = 0.99 \, . \]

The following steps are used to obtain the reliability, starting from the blocks that are composed of only either parallel edges or serial edges:

- First, combining \( e_{31} \) and \( e_{32} \) gives \( T_3 = h_{31}T_{31} + h_{32}T_{32} = 0.00002 \, , \quad \frac{h_{31}T_{31}}{T_3} = \frac{h_{32}T_{32}}{T_3} = \frac{1}{2} \); using (12) for the parallel edges \( e_{31} \) and \( e_{32} \), the flow network is reduced to Fig. 2 (b) with \( R_{31} = 0.99 \, . \) Using (13) and (14), we obtain \( H_{31} = \sum_{j=1}^{2} h_{3j} = 2 \, , \) and \( T_{31} = \frac{1}{2} \sum_{j=1}^{2} T_{3j} h_{3j} = 0.00001 \, . \)

- Using (18) for the serial connection \( e_{21} \) and \( E_{31} \) to obtain the combined edge, the flow network is reduced to Fig. 2 (c) with \( R_{21} = \min(0.9999, 0.99) = 0.99 \, . \) Using (19) and (20), we have \( T_{21} = \sum_{i=2}^{3} T_{i1} = 0.00002 \, , \) and \( H_{21} = h_{21} = H_{31} = 2 \, . \)

- Considering the parallel connection in Fig. 2 (c), \( T_2 = H_{21}T_{21} + h_{22}T_{22} = 0.00005 \, , \quad \frac{H_{21}T_{21}}{T_2} = \frac{4}{5} \, , \) and \( \frac{h_{22}T_{22}}{T_2} = \frac{1}{5} \, , \) and using (12) for the parallel connection \( E_{21} \) and \( e_{22} \), we
reduce Fig. 2 (c) to Fig. 2 (d) with \( R_{21} = \frac{0.99 \times 4}{5} + \frac{0.99 \times 1}{5} = 0.99 \). Using (13) and (14), we obtain \( H_{21} = \sum_{j=1}^{2} h_{2j} = 3 \) and \( T_{21} = \frac{1}{3} \sum_{j=1}^{3} T_{2j} h_{2j} = 0.00001 \times \frac{5}{3} \).

- Finally using (18) for serial connection \( e_1, E_{21}, \) and \( e_{41} \), we have

\[
R = \min(0.99, 0.99, 0.99) = 0.99.
\]

### 3.6 Software Failure Rate

For our analysis, the software failure rate can be obtained from the reliability assuming time dependency of the triggering conditions. Let \( T \) be the total test time. Let the software reliability be \( R \) during the total test time \( T \); thus, the software failure probability is \( (1 - R) \) during the total test time \( T \). Let \( t \) be some unit time of the operational period, for instance one year, of continuous operation. Therefore, for the unit time of operational period \( t \), the software failure rate is given by

\[
(1 - R) \frac{t}{T}.
\]

In some cases, the software is not always running. It is executed only on demand. In this case, we need to modify the definitions of \( T \) and \( t \) slightly. Let \( T \) be the total number of test runs and \( t \) be the number of estimated demands in a unit time of the operational period (i.e., one year). In such a case, (21) continues to hold.

### 4 AUTOMATED TOOL

It will be a tedious process if we directly apply the methods described in the previous sections without automated processes for software partition, data collection, edge reliability and total reliability estimations, as it is tedious to count and record all \( n_i \in N, e_y \in E, T_{ij}, T, \) and \( h_{ij} \) values manually; to update all \( q_{ij} \) and \( p_{ij} \) manually; and to estimate the overall software reliability manually using the model reduction method. However, with an automated tool, the estimation of the software reliability should be straightforward and free from human effort.

In many software development environments, the software structure, including the relationships between the calling and the called functions, is provided. For example, LabView provides this relationship in a tree structure. Therefore, it is possible, with some work, to develop a tool to generate the flow network structure. Some of the work has been done, for example [16], but extra work may be needed.
Also, a number of popular operating systems and software development environments, such as Microsoft Visual Studio and vxWorks, can select different modes, such as debug and release modes. Different modes compile and run the software differently. For example, if the debug mode is selected, it can record the execution time for any part of the software under the test. Therefore, techniques for determining the CPU times $T_{ij}$, $T$, and the number of executions $h_{ij}$ during the entire test stage are available.

We propose to add a test mode to the software development environment. It should have the following features:

1. When the software is compiled in the test mode, the tool should record nodes $n_i \in N$, edges $e_{ij} \in E$, and should create the flow network structure of the software.

2. When testing starts, in every test scenario run, the development environment should record the path tested. For each edge, $h_{ij}$ and average of $T_{ij}$ should be recorded, and $T$ should be accumulated.

3. Software test engineers are required to examine and accept/reject the test result. If the engineer accepts the test result, the development environment should update $q_{ij}$ and $p_{ij}$, according to (6-7).

4. If a software defect is identified in edge $e_{ij}$, the defect should be fixed. For all edges $e_{ij}$ involved in the fix (they may belong to different threads), $h_{ij}$ and $T_{ij}$ should be reset to zero, after which the test will continue and all edge reliability estimation processes will be updated for $q_{ij}$ and $p_{ij}$ using (6-7) as before.

5. When all testing is complete, estimate the software reliability according to (12-14) and (18-20) use the procedure presented in Sections 3.2-3.4.

6. To improve the reliability of the software, the edges tested least should undergo more tests. Therefore, the information on these edges should be provided.

7. The information on the software reliability should be kept in the release mode. It should be available for reading if a request is made.

In summary, an automated tool in the software development environment is desirable, and it should have the features listed above to facilitate software reliability estimation. We believe, with some extra effort on top of the existing software development environment, that software development tool vendors should be able to provide all of the information to assess software reliability using the proposed method.

5 CONCLUSIONS

In this paper, we proposed a systematic and probabilistic method to estimate software reliability. The method is rigorously based on test data, flow network, binomial distribution, and Bayesian estimations rather than a subjective judgement. For any components or system involving both hardware and software, the reliability estimation has been discussed in [33].
6 REFERENCES


