Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies
NPIC&HMIT 2009, Knoxville, Tennessee, April 5-9, 2009, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2009)

A FLOW NETWORK MODEL FOR SOFTWARE RELIABILITY ASSESSMENT

Yaguang Yang
United States Nuclear Regulatory Commission (see disclaimer)
Mail Stop C2A7, Washington DC, 20555-0001
yxy@nrc.gov

ABSTRACT

This paper proposes a software reliability model that is purely based on software structure and software test results. The flow network in graph theory is used to model the software structure by nodes and edges. The edges' failure probabilities are determined by the software test results. Based on the edges' failure probabilities and the flow network model, a method is proposed to calculate the software reliability. An automated tool, once developed as proposed in this paper, could be used to create the flow network model, to evaluate the failure probability for each edge, update the edges' failure probabilities during the software test stage, and calculate the software reliability. A simple example is provided to demonstrate how this method works.

Key Words: Software reliability, flow network model, probability.

1 INTRODUCTION

Software reliability has become an increasingly important issue as software intensive digital instruments and controls are replacing hardware-intensive analog instruments and controls in safety related systems in the nuclear power industry. Developing a software reliability model applicable to real world problems is more desirable than ever due to the following two reasons: first, operational safety is the paramount objective of the nuclear power industry, the Nuclear Regulatory Commission, and the public; second, there is no commonly accepted theory or practice to measure software reliability, though a lot of efforts have been made and many

research results have been published in the last three decades. For example, in a recently published NRC document [1, page 3-6] that addresses probabilistic risk assessment methods for digital systems, software failure estimation is not included because the technical community has not reached a consensus about a method to be used for this purpose.

The lack of a commonly accepted theory and practice is partially because software failures are fundamentally different from hardware failures upon which traditional reliability theory is built. For example, the primary causes of hardware failures are wear out or aging related failures. Therefore, hardware failure models are normally based on failure time that naturally fits probability distributions such as exponential distribution, Poisson distribution, Weibull distribution, and Gamma distribution, etc. Also, for the same hardware, the exact failure time of individual components are unlikely to be the same, redundancy can be used to improve the system reliability. However, software failure is due to human error rather than aging and the failure is triggered by a specific event and input data set. Therefore a software reliability model based on failure time is not appropriate. If some event(s) and input data set cause a copy of a software product to fail, the same event(s) and same input data set will cause every copy of the same software product to fail. Therefore, redundancy using copies of the same software product will not improve system reliability. This means that the traditional reliability theory may not be directly applicable to software reliability assessment. Software specific failure characteristics need to be considered while developing a reasonable software reliability model and related software reliability theory.

Influenced heavily by the traditional reliability theory that is built upon the assumptions suitable for hardware reliability applications, many approaches for software reliability are still somehow related to the failure time, these methods use past software failure data during the development and operation stages to predict future behavior of the software, i.e., reliability is considered as a function of time and time related past failures. Many published methods use either an observed number of failures in a certain time period (for example, A.L.Goel, and K. Okumoto [2]) or the time periods between software failures to predict future software reliability (for example, Jelinski and Moranda [3]). The first type of modeling uses independent Poisson random variables. Some current research still follows this direction, for example W. Wang, T. Hemminger and M. Tang [4] and references therein. The second type of modeling assumes that the software faults are fixed when they are found in the operational process, and the elapsed time between failures is taken to follow an exponential distribution. This idea remains an active research direction and recent developments consider more complicated cases, for example C. Huang and C. Lin [5] and references therein. A well written survey for these types of methods, including models using Weibull and Gamma distributions, can be found in [9]. Another widely noticed idea was proposed by Musa [6] who believed that the execution time of the software is more accurate in modeling than the operational time because the latter may include some idle time.

For time-critical software, for example, the real time software, Petri nets were adopted to model the software reliability and safety [20, 21, 22]. Similar to a flow network model, Petri nets are graphic theory models, but differ from the flow network model developed in this paper. Petri nets models use places, transitions, tokens, states, and arcs to describe the software states. The software safety or reliability is related to the reachability of certain states. The transitions among the states are the result of enabling the transitions and firing, i.e., remove a token from an input place to an output place. For each transition in a Petri nets model, there is also a transitional

probability which needs substantial information or assumptions. Therefore Petri nets model is more complicated conceptually and more difficult to implement than a flow network model.

Some recent developments consider a component-based software reliability model (Dolbec et al. [7], and Yacoub et al [8]). Again, these types of methods require transitional probability from one component to another component. Substantial operational data must be collected to determine these transitional probabilities.

A very recent effort that uses reliability metrics to estimate software reliability is developed in [10]. These models are based on reliability metrics, such as failure rate, coverage factor, cyclomatic complexity, and requirement traceability, etc and the software reliability are estimated based on these metrics. Although these metrics are intuitively related to the quality of the software, there are difficulties to build mathematical relations between the software reliability and these metrics. For example, large, complicated software may not necessarily be less reliable than small, simple software because their reliabilities also depend on other factors such as developers' ability and experience, and the efforts put into the software development.

Although software and hardware failure characteristics are totally different, a key method used in practice to gain confidence about the product reliability is the same for hardware and software, that is, "extensive testing". Hardware test results have been used, incorporate with reliability models, to estimate hardware product reliability [12]. Similarly, there are proposals to use software test results to estimate software reliability. For example, [18] describes Leone's test coverage model that estimates software reliability using a weighted average of four different coverage metrics: lines of executable code, independent test paths, functions/requirements, and hazard test cases, achieved during test phases. Though, this method uses test results, it is essentially a metric method that does not make fully use of the software structure information.

In this paper, a different modeling method is proposed. It uses a simple graphic flow network model to describe the software structure, and the reliability is then estimated based on the flow network model and test results. More specifically, the software is partitioned and represented by nodes and edges. These edges are all disjoint and all edges form the entire flow network (software). The failure probability of each edge of the flow network (software) is determined during the test stage. The entire software reliability is then calculated based on the failure probabilities of all edges of the software and the flow network model (structure of the software). There are several advantages for using this type of model. First, the modeling method and reliability analysis are suitable for assessing the software reliability before the system is operated. Second, this type of modeling technique is objective because the failure probability of each part (edge) of the software is determined by test results and the entire software reliability is determined based on both test results and flow network structure. Third, the modeling procedure makes data collection easy because the data is time independent. This is true when many developers work together at the same time to develop and test the software, and the detected fault may not be reported and/or recorded at the right time. Finally, the data collection can be straightforward and accurate if an automated tool is applied.

It will be clear in the following discussion that if a manual method is used, it can be tedious to create a flow network structure for the software under study. Also it can be time consuming to associate the test data to each edge in the flow network structure; therefore an automated tool is suggested to handle all these tedious details.

The remainder of the paper is organized as follows. Section 2 provides the relationship between software structure and the flow network model, and describes a systematic way to estimate the software reliability. It uses an example to explain the flow network modeling procedure and to show the reliability assessment steps. Section 3 proposes an automated tool to facilitate flow network modeling, data collection and updating, and reliability calculation. The conclusions are summarized in Section 4.

2 FLOW NETWORK MODEL AND RELIABILITY CALCULATION

2.1 Assumptions and Limitations

First, it is worthwhile to note that, similar to hardware reliability testing, software testing may not catch all the defects. A part of software repeatedly passing tests does not imply that the reliability of this part of the software is perfect; but its reliability can be considered high. Next, the method developed here considers purely the software reliability; it does not address related reliability issues from a system point of view. For example, software tests may not find design and/or requirements errors, though this is an important area and research is still on going, the method developed here does not intend to discuss the reliability issues caused by design and requirement errors. In the aerospace industry potential design and requirement errors are carefully examined in design reviews in various development stages and tests are used to find pure software errors. Also it is assumed in the following discussions of this paper that test scenarios mimic the environment and conditions of the software in use. Though, test design may have made every effort to meet this assumption in practice, it may not fully cover all the scenarios of the software in use. The optimization of test design is a separate issue and is not discussed in this paper. However, as it is known that test coverage and reliability estimation accuracy is closely related [19]. In the next section it is suggested that an automated tool for software reliability evaluation should provide information on the least tested edges to help add test scenarios to improve reliability estimation. Finally, the proposed method in the current form considers only single thread software. It is not clear at this time if the method can be extended to multi-tasking software or parallel software.

2.2 Flow Network Model

Unlike hardware defects, software defects are solely caused by human errors which can occur in any place in the software and at any time in the software development life cycle. To effectively calculate the software reliability based on the test results, it is desirable to know which parts of the software has been tested, how many times each part of the software has been tested, and how these parts are connected. Therefore, it is proposed to partition the software under study into small individual parts. This proposed partition satisfies the following two features. First, each part is a single thread, i.e., every line in the part will be executed if any line of the part is executed. Second, there is no intersection between any two parts, i.e., every line of the software belongs to one and only one individual part in the partition. This partition also provides the software structure information, i.e., how these parts are connected in the software. As large software may have tens of thousands or even millions of parts, it is impractical to test every part of the software during verification and validation testing for such large software. Software verification and validation tests normally run the entire software in different scenarios

to make sure that the test results in all test scenarios meet the design and test specifications. All parts of the software involved in these tests are associated with some failure probabilities depending on how many times these parts are involved in the tests. The entire software reliability is then obtained from the failure probability of every part of the software and the software structure.

Due to the features of the partition, if the software under the test has a bug, it must be in some part of the partition. Since every part is a single thread, if this part is executed, the line including the bug must be executed, and therefore, it is likely that the output of the test does not meet expectations. On the other hand, if a test scenario meets expectations, there is a lower probability of having defects in any part of the partition that was executed. Therefore a low failure probability should be assigned to all parts in the partition involved in this test scenario. We say these parts pass the test scenario. The more test scenarios a part passes, the lower the failure probability of the part should be. Some parts may never be tested in all test scenarios during the verification and validation test period; a higher failure probability should be assigned to these parts.

The above discussion suggests that the structure of flow network defined in graph theory (see [13]) is an appropriate tool to represent the partition of the software. Then each part of the software in the partition is an edge of the flow network, and the points where software branches start and end are nodes. More specifically, the flow network representing the software partition is composed of one *source*, the start point of the software or the first line inside the outmost 'while' infinite loop; one *sink*, the end of the software or the last line inside the outmost 'while' infinite loop; n *nodes* $n_i \in N$; for each node i except the sink, there are $j = 1, \cdots, j_m$ *edges* $e_{ij} \in E$ emanating from node i. Each edge in the software is a piece of the single thread software between nodes. Here single thread means that if the edge is executed, then every line inside the edge is executed, i.e., there is no branch inside an edge. It is assumed that there is 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 started with 'if' or 'switch', etc; the edges are the collections of pieces of software that meet one of the following conditions: (a) between the lines of the start of each function until meet an 'if' or 'switch'; (b) between the line '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 line 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. The following simple pseudo c/c++ code describes the partition or the flow network concept.

```
Main()                                          //node 1
{
    Data initialization;                        //edge e11
    If condition A holds                        //node 2
    {
        Process data;                           //edge e21
        If data processing success              //node 3
```

```
            {
                  Save result;                          //edge e31
            }
            Else if data processing fail
            {
                  Issue a warning;                      //edge e32
            }
      }
      Else if condition A does not hold
      {
            Print “condition A does not hold”;        //edge e22
      }                                                //node 4
      Clean memories;                                  //edge e41
}                                                      //node 5
```

By applying the principles described above, the flow network model corresponding to the pseudo code is given in Figure 1.

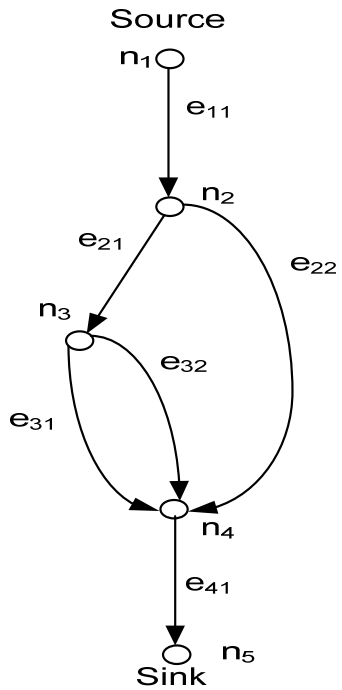

Figure1. The flow network model for simple pseudo code

A similar model (see [14]) was used to measure the cyclomatic complexity in structured testing but was not applied to the reliability problem. In [10], the cyclomatic complexity is used for measuring the software reliability, but [10] did not use test results to derive the failure

probability for each edge of the software and it did not use the structure to calculate the software reliability. In this proposed method, the flow network model is used to partition the software so that it is easy to identify which edges of the software are executed in the test runs. The test results are then used to determine the failure probabilities of each edge of the flow network. Based on the failure probabilities of all edges in the software and the flow network (software) structure, a method is proposed to determine the software reliability.

Let t_{ij} be the average execution time of the edge e_{ij} of the software and h_{ij} be the observed number of executions of e_{ij} in the software test stage. Let $e_{ij}=0$ denote the event that the edge e_{ij} has at least one defect, $e_{ij}=1$ denote the event that the edge e_{ij} has no defect, $p_{ij}=P(e_{ij}=1)$ be the probability that the edge e_{ij} has no defect and $q_{ij}=P(e_{ij}=0)=1-p_{ij}$ be the probability that the edge e_{ij} has at least one defect. If a software test scenario covers an edge e_{ij} of the software and the test results meets the expectation, then the estimated software failure probability for e_{ij} is $q_{ij}=1-p_{ij}\approx 0$ and $p_{ij}\approx 1$. Otherwise if the software test scenario does not cover e_{ik}, p_{ik} is assigned a smaller number than p_{ij} because e_{ij} passes the test and e_{ik} is not tested in the test. As previously defined $q_{ik}=1-p_{ik}$. Similarly, if an edge e_{ij} of the software passes more test scenarios than another edge e_{ik} of the software, then the failure probability of the edge e_{ij} should be smaller than the failure probability of the edge e_{ik}. In other words, if the edge e_{ij} appears more times in test scenarios than another edge e_{ik}, and all these tests meet the test expectations, the failure probability of the edge e_{ij} should be smaller than the failure probability of the edge e_{ik}, i.e., $q_{ij}<q_{ik}$ or $p_{ij}>p_{ik}$. All p_{ij} and q_{ij} should be updated as the software test proceeds. Let m_{ij} be a positive integer[1] and the failure probability of e_{ij} be $q_{ij}=10^{-m_{ij}}$ if e_{ij} is executed exactly one time and the test scenario meets the expectation. If e_{ij} is executed exactly h_{ij} times during the software test stage and all test scenarios involved in h_{ij} executions meet the expectations, the failure probability of e_{ij} is defined as[2]

$$q_{ij}=10^{-h_{ij}m_{ij}}. \tag{1}$$

Hence if e_{ij} is executed exactly h_{ij} times in the software test stage and all h_{ij} executions meet the expectations, then

$$p_{ij}=1-10^{-h_{ij}m_{ij}}. \tag{2}$$

Therefore, at the end of the test stage, the software reliability can be modeled by the flow network with nodes and edges, each edge has a reliability determined by the test results, and the

[1] m_{ij} can be a function of failure metrics such as line of codes, developers experience and past performance, etc. In the example introduced later, m_{ij} is assumed to be a constant to make the problem simple.

[2] A general failure probability function $f(h_{ij},m_{ij})$ can be defined as long as it meets the condition that the more test scenarios the edge passes (larger h_{ij}), the lower failure probability the edge will have.

connections of nodes and edges are represented by the structure of the flow network. Clearly, the success and failure probability of each edge e_{ij} in a single execution is a *Bernoulli* distribution. The success and failure probability of each edge e_{ij} in h_{ij} executions is a *binomial* distribution. Based on this observation, a software reliability model can easily be simplified according to how these edges are connected. In the following discussions, reliability calculations for parallel and serial connections are presented separately.

2.3 Reliability Calculation for Parallel Connected Edges

First, for a block under node i composed of j_m parallel connected edges, the total number of executions of all parallel connected edges e_{ij} under node n_i during the test stage is $h_i = \sum_{j=1}^{j_m} h_{ij}$, and the total execution time of all parallel connected edges e_{ij} under node n_i during the test stage is $t_i = \sum_{j=1}^{j_m} t_{ij} h_{ij}$.

Since e_{ij} has h_{ij} number of executions in the test stage, and the probability of each edge tested successfully is a *binomial* distribution, for all parallel connected edges immediately under node i, the following relation is held

$$\sum_{j=1}^{j_m} \frac{h_{ij} t_{ij}}{t_i} (p_{ij} + q_{ij})^{h_{ij}} = 1. \tag{3}$$

Therefore, the reliability of the block composed of parallel connected edges under node i in the software is given by

$$R_{i1} = \sum_{j=1}^{j_m} \frac{h_{ij} t_{ij}}{t_i} p_{ij}^{h_{ij}} \tag{4}$$

The execution time for the combined artificial edge (from the parallel block) used in the next calculation is

$$t_{i1} = \frac{1}{H_{i1}} \sum_{j=1}^{j_m} t_{ij} h_{ij}. \tag{5}$$

The number of executions of the combined artificial edge used in the next calculation is taken as the summation of the number of executions of all the edges in the parallel block

$$H_{i1} = \sum_{j=1}^{j_m} h_{ij}. \tag{6}$$

The same method can be applied to the parallel connected blocks if each block is reduced to an artificial edge.

2.4 Reliability Calculation for Serial Connected Edges

For a block under node i_1 composed of nodes $i_1 \ldots i_s$ and serial connected edges (there are no parallel connected edges in all nodes $i_1 \ldots i_s$), the total number of executions of all serial connected edges e_{ij} under node n_{i_1} during the test stage is $h_i = h_{ij}$ for any $i \in i_1 \ldots i_s$, and the total execution time of all serial connected edges e_{ij} under node n_i during the test stage is $t_i = \sum_{i=i_1}^{i_s} t_{ij} h_{ij}$.

Since e_{ij} has h_{ij} number of executions in the test stage, and each edge has *binomial* distribution, for all serial connected edges immediately under node i, the following relation is held

$$\prod_{i=i_1}^{i_s} (p_{ij} + q_{ij})^{h_{ij}} = 1. \tag{7}$$

The reliability of the block composed of serial connected edges under node i_1 in the software is given by

$$R_{i_1 j} = \prod_{i=i_1}^{i_s} p_{ij}^{\,h_{ij}} \tag{8}$$

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

$$t_{i_1 j} = \sum_{i=i_1}^{i_s} t_{ij} \,. \tag{9}$$

The number of executions of the combined artificial edge is taken as the number of executions of any edge h_{ij}, where $i \in i_1 \ldots i_s$, in the serial block

$$H_{i_1 j} = h_{ij} \,. \tag{10}$$

The same method can be applied to the serial connected blocks if each block is reduced to an artificial edge.

When (4) and (8) are used to simplify the flow network, several edges e_{ij} are combined into an artificial edge, which is denoted by E_{lk} with l the smallest index among i's and k the smallest index among j's in the edges of e_{ij}. The software reliability is obtained by continuing the process until all the edges have been combined into a single artificial edge.

2.5 An example

The pseudo c/c++ code example introduced previously is used to demonstrate how this method of calculating software reliability works. The software partitioned as in Figure 2 (a) has five nodes and six edges. Assume also that three tests are conducted. The first test path is $e_{11}e_{21}e_{31}e_{41}$, the second test path is $e_{11}e_{21}e_{32}e_{41}$, and the third test path is $e_{11}e_{22}e_{41}$. Assume further that the total test time is $t = .00011$ hours and $t_{ij} = .00001$ hours for every edge. Therefore, it has $h_{11} = h_{41} = 3$, $h_{21} = 2$, and $h_{22} = h_{31} = h_{32} = 1$. Let $m_{ij} = 2$ for all edges, then

$p_{11} = p_{41} = 0.999999$, $p_{31} = p_{32} = p_{22} = 0.99$, $p_{21} = 0.9999$. 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, notice that $t_3 = h_{31}t_{31} + h_{32}t_{32} = 0.00002$, $\frac{h_{31}t_{31}}{t_3} = \frac{h_{32}t_{32}}{t_3} = \frac{1}{2}$, then use (4) for the parallel edges e_{31} and e_{32}, the flow network is reduced to Figure 2 (b) with $R_{31} = 0.99$. Using (5) and (6), we obtain $t_{31} = \frac{1}{2}\sum_{j=1}^{2} t_{3j}h_{3j} = 0.00001$ and $H_{31} = \sum_{j=1}^{2} h_{3j} = 2$.

- Using (8) for the serial connection e_{21} and E_{31}, the flow network is reduced to Figure 2 (c) with $R_{21} = 0.9999^2 * 0.99^2$. Using (9) and (10), we have $t_{21} = \sum_{i=2}^{3} t_{i1} = 0.00002$ and $H_{21} = h_{21} = H_{31} = 2$.

- Use the fact that $t_2 = H_{21}t_{21} + h_{22}t_{22} = 0.00005$, $\frac{H_{21}t_{21}}{t_2} = \frac{4}{5}$, $\frac{h_{22}t_{22}}{t_2} = \frac{1}{5}$, and use (4) for the parallel connection E_{21} and e_{22}, the flow network is reduced to Figure 2 (d) with $R_{21} = \frac{0.9999^2 * 0.99^2 * 4}{5} + \frac{0.99 * 1}{5} = p_{21}^3$. Use (5) and (6), we obtain $t_{21} = \frac{1}{3}\sum_{j=1}^{2} t_{2j}h_{2j} = 0.00001 * \frac{5}{3}$ and $H_{21} = \sum_{j=1}^{2} h_{2j} = 3$.

- Finally using (8) for serial connection e_1, E_{21}, and e_{41}, the software reliability is given by

$$R = 0.999999^3 * \left(\frac{0.9999^2 * 0.99^2 * 4}{5} + \frac{0.99 * 1}{5} \right) * 0.999999^3 = 0.981917.$$

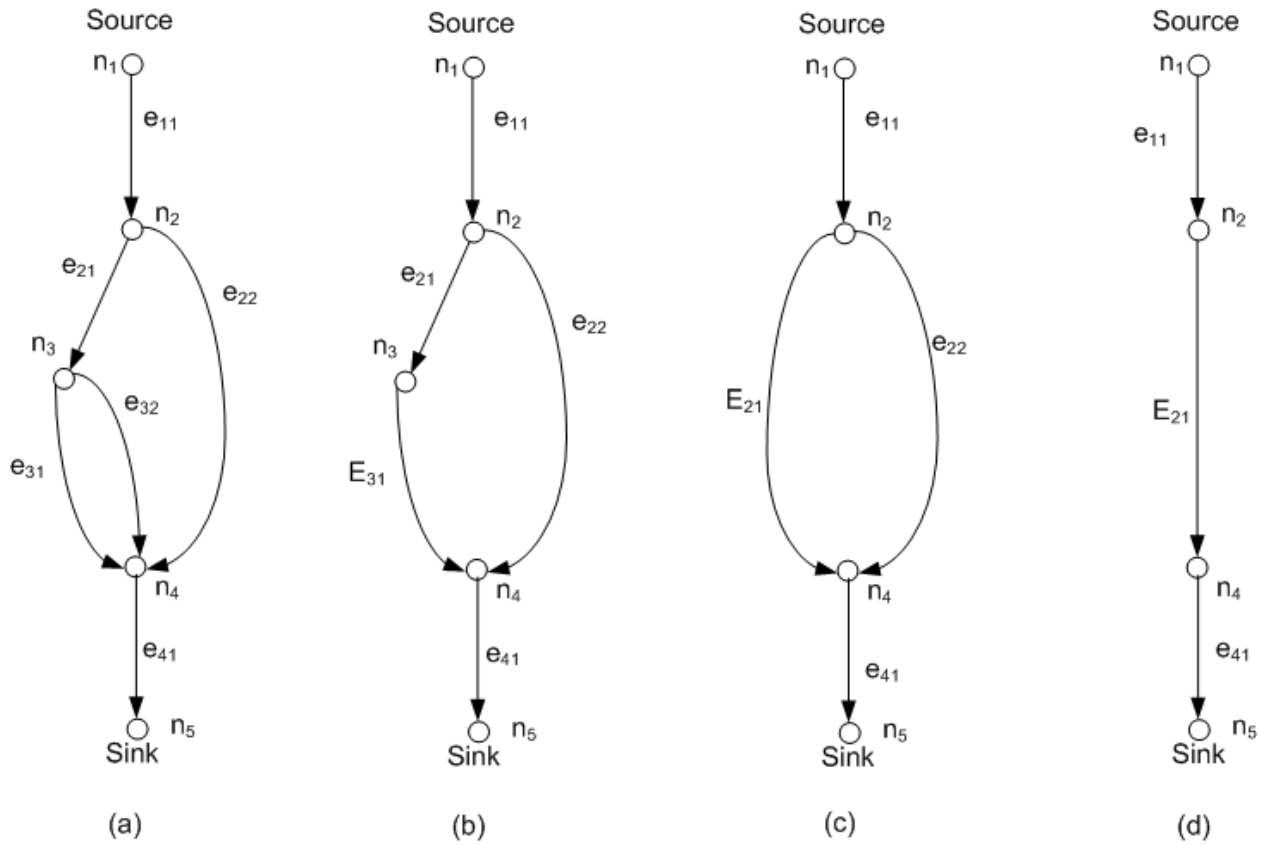

Figure 2. Reliability Calculation Procedures

2.6 Failure Rate

For some practical applications, software failure rate other than software reliability is required. This can easily be obtained from the reliability. Let T be the unit time period, say per year, of continuous operation. The software failure rate, given the software reliability, is given by

$$(1-R)*\frac{T}{t}. \tag{11}$$

The steps of implementing the above described method are summarized as follows:

1. Construct the flow network that is composed of nodes $n_i \in N$ and edges $e_{ij} \in E$.
2. Record t_{ij}, t, h_{ij}, and update q_{ij} and p_{ij} during the test stage. If a bug is found in edge e_{ij}, simply fix the bug, reset $h_{ij} = 0.5$, reset q_{ij} and p_{ij} according to (1) and (2), keep values for the rest h_{ij}, q_{ij} and p_{ij}, and then continue the test.
3. When all tests are done and no defect is detected, calculate the software reliability R by repeatedly using formula (4) (5) (6) for parallel connections and (8) (9) (10) for serial connections until the solution is reached.

Clearly the above manual process can be very tedious for large software packages. Therefore it is desirable to develop an automated tool to create the model, collect the data, and calculate the reliability. This is the topic of the next section.

3 PROPOSED AUTOMATED TOOL

As mentioned above, it will be difficult to implement the methods described in the previous section without an automated process for software partition, data collection and failure probability update, and calculation of the reliability, because it is very tedious to manually count and record all $n_i \in N$, $e_{ij} \in E$, t_{ij}, t, and h_{ij}; manually update all q_{ij} and p_{ij}; and manually calculate the entire software reliability. However with an automated tool, the calculation of the software reliability should be straightforward.

In many popular software development tools (including compilers), the software structure, including the relations between the calling and the called functions, is provided. For example, LabView provides this relationship in a tree structure. This means that it is possible, with some extra work, to develop a tool to generate the flow network structure so that this tool can provide the information on $n_i \in N$, $e_{ij} \in E$,, and how these edges and nodes are connected.

Also, some popular operating systems and software development tools, such as Microsoft Visual Studio and vxWorks, can set options for different modes such as debug and release modes. Different modes compile and run the software differently. For example, if debug mode is selected, it can record the execution time for any part of the test software. Therefore, techniques to record CPU time t_{ij}, t, and the number of executions h_{ij} during the entire test stage are available.

It is proposed to add another mode, test mode, into software development tools. When this mode is selected, it should have the following functions.

1. When the software is ready for the test, the software is compiled in this mode. The tool should record *nodes* $n_i \in N$, all distinct *edges* $e_{ij} \in E$, and create the flow network structure of the software. It should also set points where t_{ij} and h_{ij} will be recorded when edge e_{ij} is executed.
2. When software test starts (mode has been set to test mode), in every test scenario run, the development tool should record h_{ij}, take average of t_{ij}, and accumulate t.
3. If the software engineers examined and accepted the test result, the development tool should update q_{ij} and p_{ij}, according to (2) and (3), and calculate software reliability based on (4) and (8).
4. If a software defect is identified in an edge e_{ij}, the defect should be fixed, the h_{ij} should be reset to one half, q_{ij} and p_{ij} should be reset according to the new h_{ij}, the test will continue, and the reliability calculation will be updated as before. This means that reliability is only considered for software that does not have any known defects at the end of the test stage, though the software may still have some unidentified defects.
5. To improve software reliability, the least tested edges should have more tests. Therefore, the information on least tested edges should be provided.
6. The information on the software reliability should be kept in release mode.

In summary, an automated tool in the software development environment is desirable and it should have the features listed above to facilitate software reliability calculations. It is believed, with some extra efforts on top of the existing software development environment, the software development tool vendors should be able to provide all the information to assess software reliability.

4 CONCLUSIONS

In this paper, an objective method to evaluate the software reliability based on the software structure and software test results is proposed. The flow network model is adapted to model software structure. Since software failures are human errors that can happen in any part of the software, it is assumed that the probability to catch these human errors is high if the lines including these errors are executed in the software test and test results are carefully examined. An automated tool is proposed to create the flow network model and identify what parts (edges) of the software have been tested. For each tested part (edge) of the software, using the facts that the pass/fail outcome in a single execution follows Bernoulli distribution and the pass/fail outcome in multiple executions follows binomial distribution, the software reliability is derived as the function of the failure probabilities of all the edges and the software structure.

5 ACKNOWLEDGMENTS

The author would like to thank Mr. Russell Sydnor (Office of Nuclear Reactor Research, NRC) for his encouragement, support, and many detailed comments; Mr. Steven Arndt (Office of Reactor Regulation, NRC), Ms. Debra S. Herrman (Office of New Reactor, NRC), and Dr. Y. James Chang and Mr. Daniel Santos (Office of Nuclear Reactor Research, NRC) for their invaluable comments and useful discussions. Their inputs helped to improve the quality of the paper.

6 REFERENCES

1. T.L. Chu, G. Martinez-Guridi, M. Yue, J. Lehner, and P. Samanta, “Traditional Probabilistic Risk Assessment Methods for Digital Systems”, NUREG/CR-6962, US Nuclear Regulatory Commission, October, 2008.
2. A.L.Goel, and K. Okumoto, “Time-Dependent Error -Detection Rate Model for software and Other Performance Measures,” IEEE Transactions on Reliability, **Vol. R-28**, no. 8, August 1979, pp. 206-211.
3. Z. Jelinski, and P.B. Moranda, Software Reliability Research, Proceedings of the Statistical Methods for the Evaluation of Computer System Performance, Academic Press, 1972, pp. 465-484.
4. W. Wang, T. Hemminger, and M. Tang, “A moving average Non-Homogeneous Poisson Process Reliability Growth Model to Account for Software with Repair and System Structure,” IEEE Trans. On Reliability, **Vol. 56**, No. 3 pp. 411-421, 2007.
5. C. Huang and C. Lin, “Software Reliability Analysis by Considering Fault Dependency and Debugging Time Lag,” IEEE On Reliability, **Vol. 55**, No. 2 pp. 436-450, 2006.

6. J. D. Musa, A. Iannino, and K. Okumoto, Software Reliability Measurement, Prediction, and Applications, New York: McGraw-Hill, 1987.
7. J. Dolbec and T. Shepard, "A Component Based Software Reliability Model," Proc. Conference for Advanced Studies on Collaborative Research, Toronto, Canada, 1995.
8. S. Yacoub, B. Cukic, and H.H. Ammar, "A scenario-based reliability analysis approach for component-based software," IEEE Transactions on Reliability, **Vol. 53**. No. 4, pp. 465- 480. 2004.
9. W. Farr, "Software Reliability Modeling Survey," in Handbook of Software Reliability Engineering, Edited by Michael R. Lyu, IEEE Computer Society Press and McGraw-Hill Book Company, 1996.
10. C. S. Smidts, M. Li, W. Kong, Y. Shi, and S. Ghose, "Large Scale Validation of a methodology for Assessing Software Quality," Research Report Submitted to Nuclear Regulatory Commission, Rockville, Maryland, 2007.
11. Martin A. Stutzke, and C. S. Smidts, "A Stochastic Model of Fault Introduction and Removal During Software Development," IEEE Transactions on Reliability, **Vol. 50**, No. 2, pp. 184-193, 2001.
12. W. Q. Meeker and L. A. Escobar, Statistical Methods for Reliability Data, Wiley Series in Probability and Statistics, 1998.
13. "Flow Network," http://en.wikipedia.org/wiki/Network_flow.
14. A. H. Watson, T. J. McCabe, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-235, 1996, http://www.mccabe.com/pdf/nist235r.pdf.
15. K. Malaya, M. Li, and J. Bateman, and R. Karmic, "Software Reliability Growth with Test Coverage," IEEE Transactions on Reliability, **Vol. 51**, No. 4 pp. 420-460, 2002.
16. R. C. Cheung, "A User-Oriented Software Reliability Model," IEEE Trans. On Software Engineering, **Vol. 6**, No. 2, pp. 118-125, 1980.
17. Igor Basov's, Reliability Theory and Practice, International series in Engineering, Prentice-Hall Inc. 1961.
18. A. Neufelder, Ensuring Software Reliability, Marcel Dekker Inc., 1993, pp. 137-140.
19. Yashwant K. Malaiya, et al., The relationship between test coverage and reliability, in Proceedings of International Symposium on Software Reliability Engineering, Nov. 1994, pp. 186–195.
20. Debra S. Herrmann, Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors, IEEE Computer Society, 1999.
21. Ugo Buy and Robert H. Sloan, Analysis of Real-Time Programs with Simple Time Petri Nets, in Proceedings of the 1994 International Symposium on Software Testing and Analysis, pp.228-239, August, 1994.
22. Nancy G. Leveson and Janice L. Stolzy, Safety Analysis Using Petri Nets, IEEE Transactions on Software Engineering, **Vol. 13**, No. 3, 1987.