http://dx.doi.org/10.5516/NET.04.2012.513

RELIABILITY ESTIMATION FOR A DIGITAL INSTRUMENT AND CONTROL SYSTEM

YANG YAGUANG[1*] and SYDNOR RUSSELL[2]
[1]Office of Research, Nuclear Regulatory Commission, Rockville, 20850, USA
[2]Office of Research, Nuclear Regulatory Commission, Rockville, 20850, USA
[*]Corresponding author. E-mail : yaguang.yang@nrc.gov

Invited September14, 2011
Received September 07, 2011
Accepted for Publication March 10, 2012

In this paper, we propose a reliability estimation method for DI&C systems. At the system level, a fault tree model is suggested and Boolean algebra is used to obtain the minimal cut sets. At the component level, an exponential distribution is used to model hardware failures, and Bayesian estimation is suggested to estimate the failure rate. Additionally, a binomial distribution is used to model software failures, and a recently developed software reliability estimation method is suggested to estimate the software failure rate. The overall system reliability is then estimated based on minimal cut sets, hardware failure rates and software failure rates.

KEYWORDS : Reliability, DI&C, Software Reliability, Fault Tree

1. INTRODUCTION

Digital instrumentation and control (DI&C) systems are widely adopted in various industries, as their flexibility and ability to implement various functions can be used to monitor, analyze, and control complex systems automatically. It is anticipated that DI&C systems will replace the traditional analog instrumentation and control (AI&C) systems in future nuclear reactor designs. There is increasing interest in reliability and risk analyses for safety-critical DI&C systems in regulatory organizations, such as The United States Nuclear Regulatory Commission.

Developing reliability models and reliability estimation methods for digital control and protection systems will involve every part of the DI&C system, such as the sensors, signal conditioning and processing components, transmission lines and digital communication systems, (digital to analog) D/A and (analog to digital) A/D converters, computer system, signal processing software, control and protection software, power supply system, and actuators. Some of these components are analog hardware, such as the sensors and actuators. Their failure mechanisms are well understood, and the traditional reliability model and estimation methods can be directly applied. However, many of these components include (a) digital hardware, such as boards, cards, or FPGAs (field-programmable gate arrays) which contain digital gates and/or memory composed of millions of transistors, capacitors, resistors, and associated communication links; (b) firmware which has software embedded into the digital hardware so that it can provide complex functions as desired; (c) system software such as Unix, Linux or Windows which coordinates the work of a system composed of digital hardware and firmware (boards and cards), and provides an interface for application users; and (d) application software which monitors (via sensors) and controls (via actuators) safety-related and non-safety-related nuclear power plant systems. To estimate the realibility of such a large system is challenging. In particular, the software needs special consideration because its failure mechanism is unique; the reliability estimation method for a software system should be different from that used for hardware.

Owing to regulatory needs in the nuclear industry and the technical challenges, many attempts have been made to find practical ways to improve the software reliability and to estimate the reliability. Dennis Lawrence [1] discussed activities that should be carried out throughout the software life cycle. Parnas, Asmis, and Madey [2] emphasized documentation requirements and quality control, including testing and reviews. Leveson and Harvey [3] proposed software fault tree analysis method. However, the most extensively investigated software reliability method is the software reliability growth model [4]. Many statistics models, such as the exponential distribution, Poisson distribution, Weibell distribution, and Gamma distribution models have been considered, and many different scenarios have been discussed [5-8]. Nonetheless, there is no formed consensus on an ideal software reliability estimation method.

At the system level, several probabilistic reliability analysis models of DI&C systems are discussed in IEC standard 61508 [9]. One of the widely used models in the nuclear industry is the fault tree/event tree model, which was first conceived by Watson [10] in 1961. This method was discussed in detail in [11]. It has had many applications, including some recent attempts to demonstrate a PRA (probabilistic risk analysis) for a digital instrumentation and control system [12] and for an ESBWR (economic simplified boiling water reactor) PRA analysis described in the ESBWR certification PRA document [13]. Because there is no consensus on a method of software reliability estimation despite the great efforts expended over the past few decades, software reliability is not addressed in [12] and software reliability is assumed to be some constant in [13], both of which are less than ideal.

In this paper, we propose a reliability estimation method for DI&C systems using a recently developed software reliability estimation method and a traditional hardware reliability estimation method. For the purpose of completeness, we will briefly describe the traditional hardware reliability estimation method so that we can show how this method is incorporated into the probabilistic reliability analysis of DI&C systems. Our main focus, however, is our own idea on how to model software failures and how to estimate software reliability based on the software failure model and test results, as well as how to evaluate overall the reliability of the DI&C systems.

The remainder of the paper is organized as follows. Section 2 presents the fault tree reliability model for DI&C systems and explains the modeling procedures using a simple artificial example. Section 3 briefly discusses a hardware reliability model and failure rate estimation method. Section 4 introduces a software reliability model and discusses in detail the software failure rate estimation method. Section 5 provides a reliability model and failure rate estimation method for firmware, which has software embedded into hardware. Section 6 presents the estimation method to determine the overall DI&C system failure rate. The last section gives the conclusions of this paper.

2. SYSTEM RELIABILITY MODEL

In this section, we discuss how the fault tree method is applied to create a full reliability model of a digital instrumentation and control system, which includes both software and hardware.

The first consideration in building such a system reliability model is the level of detail that is needed in this reliability model. In theory, the more details the model has, the higher the level of fidelity the model will exhibit. However, this may not be realistic. For example, digital circuit boards can have millions of transistors. If we include all of these transistors, the model will be too complicated. Given that acceptance tests and pass/fail decisions are most likely conducted at the board level, we suggest that the level of detail should not be deeper than the board level. All of the hardware and software should be included, but the hardware and software should be considered separately because the failure mechanism is different and the reliability models are different, as we will discuss later.

Second, the model should be system specific, i.e., it should depend on a specific DI&C system design. For illustration purposes, we use a simple artificial example to describe the modeling procedure. Fig. 1 is a simplified DI&C system which has three identical redundant smart sensors which have both hardware and software. The measurements from the three sensors are sent to an A/D converter, the signal is processed in a single-board computer, and the control command is then sent to a D/A converter and then to an actuator. All components have two different failures, i.e., aging-related failures and physical-damage-related failures, except for the single-board computer and the smart sensors, which have two failure modes, i.e., hardware failure and software failure. We also assume that the A/D always receives signals (correct or incorrect) from the three sensors while the signals are useful only if two of the sensors provide correct measurement.

Following the convention used in earlier work [11], we use the concept of unreliability in the remaining

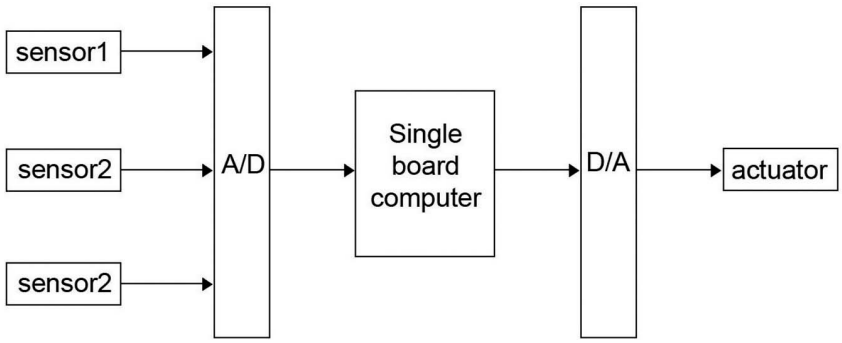

Fig. 1. A Simple DI&C System

discussion. The fault tree can then be created. It is shown in Fig. 2.

Here, A, C, E, G, K, and M are aging-related failures; H, L, and N are physical-damage-related failures, I denotes computer hardware failures, and B, D, and F are software failures due to common-cause failure events X in smart sensors. J denotes software failures in the computer. Let "+" denote the logic "or" and "•" denote the logic "and." Using the symbols defined in Fig. 2, the fault tree model can be converted to Boolean equations, as follows:

$$S = M + N + DA$$
$$DA = K + L + P$$
$$P = I + J + AD$$
$$AD = H + G + S1 \bullet S2 + S2 \bullet S3 + S1 \bullet S3 + S1 \bullet S2 \bullet S3$$
$$S1 = A + B = A + X$$
$$S2 = C + D = C + X$$
$$S3 = E + F = E + X$$

Boolean algebra can be used to reduce the Boolean equations into equivalent minimal cut sets which define the "failure modes" of the DI&C failure events. This gives

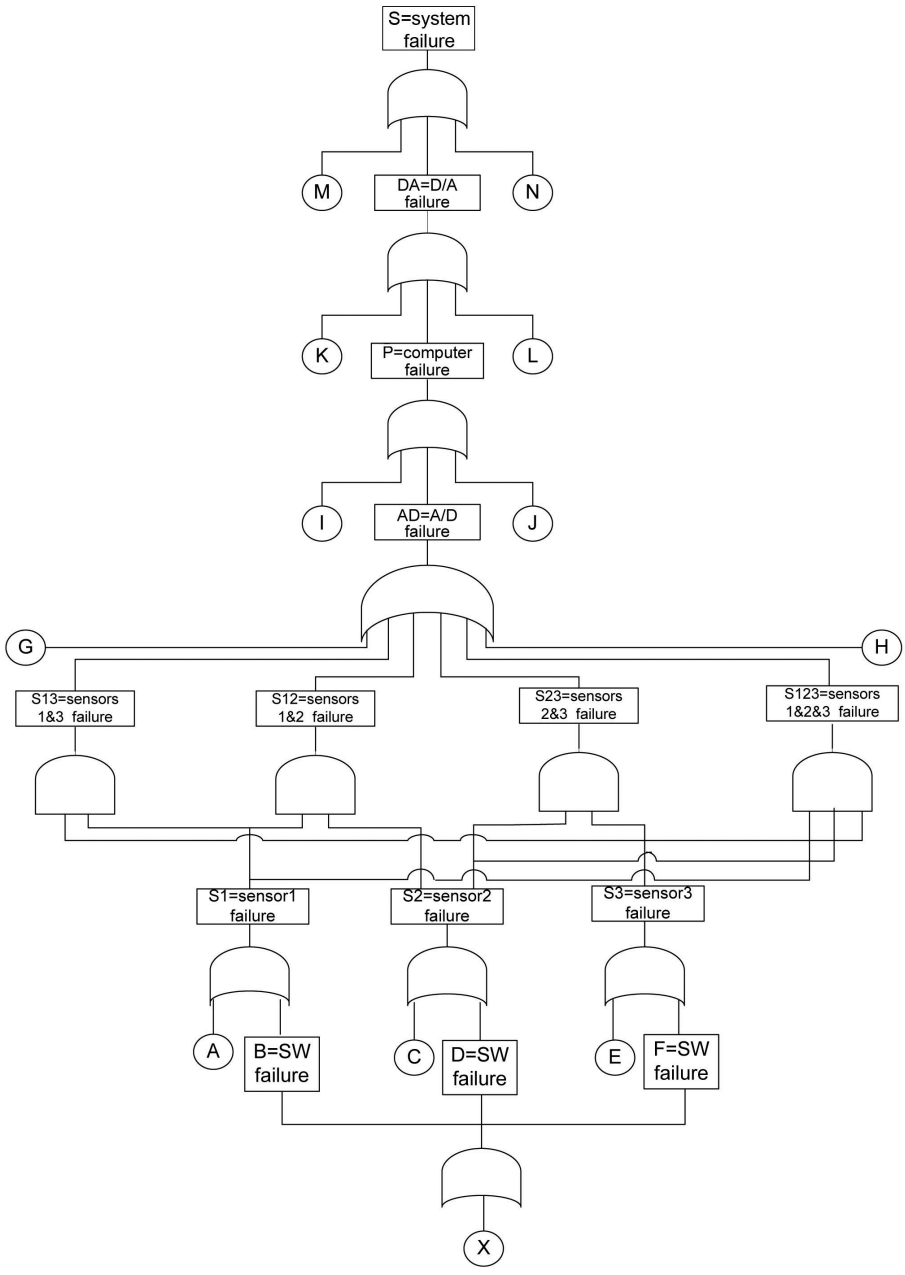

Fig. 2. Fault Tree of the DI&C System

$S = M + N + DA = N + M + L + K + P$
$= N + M + L + K + J + I + G + H + S1 \bullet S2 + S2 \bullet S3 + S1 \bullet S3 + S1 \bullet S2 \bullet S3$
$= N + M + L + K + J + I + G + H + (A + X) \bullet (C + X) + (C + X) \bullet (E + X)$
$+ (A + X) \bullet (E + X) + (A + X) \bullet (C + X) \bullet (E + X)$
$= N + M + L + K + J + I + G + H + A \bullet C + C \bullet E + A \bullet E + X$

where the last expression represents the minimal cut sets. From the minimal cut sets, it becomes clear that the system failure can be caused by single failures, G, H, I, J, K, L, M, N, and X; and by double failures, (A • C), (A • E), (C • E). The evaluation of the minimal cut set (failure mode) probabilities and the system failure probability can be straightforward if the component failure probabilities are obtained. It should be noted that this modeling method does include a specific common-cause failure, i.e., when B, D, and F, the software in smart sensors, fail at the same time due to common-cause failure X.

More details on fault tree models can be found in the literature [11] and one can always follow the standard given in earlier work [9]. Very detailed examples [11] can also be accessed while creating a unique fault tree model. In the remainder of the paper, we assume that a fault tree for a specific DI&C system has been created, that minimal cut sets were obtained using Boolean algebra, and that single failure paths and multiple failure paths were given by the minimal cut sets. We will focus on the details of how to obtain the hardware failure rates, software failure rates, firmware failure rates, and how to use the failure rates and minimal cut sets to estimate the overall DI&C system reliability.

3. HARDWARE COMPONENT FAILURE MODEL

3.1 Hardware Probabilistic Failure Model

In the aforementioned study [11], the model for the constant failure rate per hour, which has an exponential distribution, is suggested as a component failure model. The advantage of this model is its simplicity and the fact that it characterizes the main feature of hardware failures; i.e., the failure probability increases over time. For this model, the probability $F(t)$ that the component experiences its first failure within time period t, given it is initially operation, is

$$F(t) = 1 - e^{-\lambda t}. \quad (1)$$

The reliability $R(t)$ is given by the following equation:

$$R(t) = 1 - F(t) = e^{-\lambda t} \quad (2)$$

The density of the exponential distribution $f(t)$ is given by

$$f(t) = \lambda e^{-\lambda t}. \quad (3)$$

More complicated models, such as the Weibull and Gamma failure distribution models can be used in a similar way [14], but they are not discussed in this paper because, for the fault tree model, the exponential distribution is adequate [11, XI-10].

3.2 Hardware Failure Rate Estimation

For an exponential distribution, the failure rate λ for different components can be found in various documents, such as [15-16]. However, most failure rate data collections are either too old or too small for real applications, and many do not provide details on how the data were collected and calculated [12]. Another means of obtaining the failure rate information is from hardware vendors, as they normally conduct reliability tests of their products and may have reasonably accurate failure rate estimations for their specific pieces of equipment. We recommend here a Bayesian estimation method which can be used to estimate the hardware failure rate either by vendors, system integrators, or by regulatory staff members. Let $g(\lambda)$ be a priori distribution of the failure rate λ. Let n be the number of the total tested hardware components and t_i be the time when the i th hardware component fails in the test. Therefore, the average time for the hardware to fail in the test is

$$\bar{t} = \frac{1}{n}\sum_{i=1}^{n} t_i. \quad (4)$$

Using Bayesian principles, the posterior density of λ is sourced from earlier work, as follows [17]:

$$p(\lambda \mid t_1,...,t_n) = \frac{p(t_1,...,t_n \mid \lambda) g(\lambda)}{\int p(t_1,...,t_n \mid \lambda) g(\lambda) d\lambda}. \quad (5)$$

For the sake of algebraic convenience, a conjugate priori of gamma distribution is suggested for λ, which results in

$$g(\lambda : \alpha, \beta) = \frac{\beta^{\alpha}}{\Gamma(\alpha)} \lambda^{\alpha-1} e^{-\lambda\beta}, \quad (6)$$

where α can be interpreted as the number of priori observations and β as the sum of the prior observations. The posterior density has the form of a gamma distribution, as follows (see [18]):

$$p(\lambda) = g(\lambda : \alpha + n, \beta + n\bar{t}). \quad (7)$$

The estimated failure rate $\bar{\lambda}$ is the mean of (7), i.e.,

$$\bar{\lambda} = \frac{\alpha + n}{\beta + n\bar{t}}. \quad (8)$$

If $\alpha = 0.5$, $\beta = 0$, and $n = 1$, (8) is the same as the formula given in earlier work [19]. However, we believe that (8) is a better formula for general cases because it results from the assumption that the failure rate satisfies an exponential distribution.

4. SOFTWARE COMPONENT FAILURE MODEL

Software failures are fundamentally different from

hardware failures. Typical hardware failures are due to wear out or aging-related failures. Therefore, hardware failure models are based on a random failure time that can be described by exponential, Poisson, Weibull, or Gamma distributions, for example. 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 failure occurs when triggering events direct software to execute a problematic part of the software and a triggering data set is in use. Therefore, software failure models based on a random failure time may be inappropriate. Instead, we should consider software-specific failure characteristics while developing a useful software reliability model and introducing a software reliability estimation method. In particular, we model the software failure probability using a binomial distribution. If a piece of software contains some error(s), then there is a probability that the software with error(s) will fail if some triggering event occurs and if a triggering data set is in use. Moreover, a test will catch the failure when this occurs. The software failure rate is then introduced according to the ratio of the execution time of distributions faulty software that causes failure(s) and the total execution time.

Though there is no consensus on a method to be used for software reliability assessments, we believe that a recently developed test-based method is well suited for software reliability estimations [20].

4.1 Flow Network Model of Software

It was suggested in the referenced work [20] that the structure of the software should be taken into consideration in software reliability assessments 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 [21] because more information is used in the estimation. To simplify our presentation and save space, we focus on single-thread software. For a multi-thread case, we refer the readers to the literature [22].

We use a flow network 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* $e_{ij} \in E$, $j = 1, \cdots, j_m$ 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(){                                  //node 1
Data initialization;                     //edge e11
    If condition A holds                 //node 2
    {
       Process data;                     //edge e21
       If data process success           //node 3
       {
          Save result;                   //edge e31
       }
       Else if data process fail
       {
          Issue a warning;               //edge e32
       }
    }
    Else if condition A does not hold
    {
          Print "condition fail";        //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 Fig. 3.

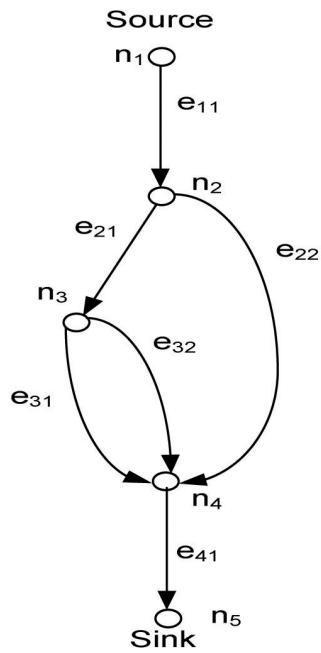

Fig. 3. The Flow Network Model of the Pseudo Code

4.2 Reliability Estimation for a Single Edge

Let T_{ij} be the average execution time of the edge e_{ij} of the software and h_{ij} be the total number of executions of e_{ij} in the software test stage. If an edge e_{ij} of the software is executed in a test scenario, we consider that the test scenario covers the edge e_{ij}. Let $e_{ij} = 1$ denote an event in which edge e_{ij} has been executed once and where the test result meets the expectations, and let $p_{ij} = P(e_{ij} = 1)$ denote the probability of this event occurring. Let $e_{ij} = 0$ denote an event in which edge e_{ij} has been executed once and where the result fails to meet the expectations, and let $q_{ij} = P(e_{ij} = 0) = 1 - p_{ij}$ be the probability of this event occurring. Therefore, p_{ij} is the probability of having no error in e_{ij} and q_{ij} is the probability of having at least one error in e_{ij}. Clearly, the one-time test scenario follows a Bernoulli distribution. Let m_{ij} be a positive real number[1]. We can set the failure probability of e_{ij} to $q_{ij} = 10^{-m_{ij}}$ if e_{ij} is executed exactly one time and the test result meets the expectation.

If all software test scenarios at the test stage do not cover e_{ik} but cover instead e_{ij}, p_{ik} should be assigned a smaller number than p_{ij}, or equivalently, q_{ik} should be assigned a larger number than q_{ij} because e_{ij} passes some tests and e_{ik} is not tested. For example, we may choose

$$q_{ik} = 10^{-\frac{m_{ik}}{2}}. \tag{9a}$$

As previously described, we define $p_{ik} = 1 - q_{ik}$.

It is likely that some edges are executed more than once in the software test stage. These multiple-test scenarios follow a binomial distribution. Also, some edges may be tested more times than other edges. A main question in a multiple-test scenario is how to determine failure probability or reliability in these situations. The following reasoning is proposed. First, if the software test scenarios cover edge e_{ij} of the software multiple times and the test results meet the expectations, then the estimated software failure probability for e_{ij} should be reduced as more tests meet the expectations. Moreover, if edge e_{ik} is executed more times than edge e_{ij} in the test scenarios and if all of these tests involving e_{ik} meet the test expectations, the failure probability of edge e_{ik} should be smaller than the failure probability of edge e_{ij}, i.e., $q_{ij} > q_{ik}$ or $p_{ij} < p_{ik}$. Therefore, if e_{ij} is executed exactly h_{ij} times and all tests meet the expectations during the software test stage, the failure probability of e_{ij} is defined as[2]

$$q_{ij} = 10^{-h_{ij} m_{ij}}, \tag{9b}$$

where m_{ij} can be any positive real number. If e_{ij} passed previous (h_{ij} -1) tests and all of the tests met the expectations but the h_{ij} th test fails to meet the expectations, the problem in edge e_{ij} must be resolved. Because the code of e_{ij} is changed after the modification, it follows a different binomial distribution. As we have not tested this newly modified e_{ij} thus far, we reset $h_{ij} = 0$ and consider the failure probability as (9a).

We assume that the failure probability is reduced exponentially with the number of continuously successful tests, as each edge is very simple (without logic branches) and because the number of lines of code in an edge is normally small. This assumption has not been validated thus far and is the main weakness of the suggested method for now. We plan future work on a rigorous statistical estimation of q_{ij}.

As the software test proceeds, all p_{ij} and q_{ij} values should be updated using (9), and

$$p_{ij} = 1 - q_{ij}. \tag{10}$$

Because multiple-test scenarios follow a binomial distribution, if e_{ij} is executed exactly h_{ij} times, the failure probability of e_{ij} includes the probability that all h_{ij} tests fail to meet the expectations, the probability that $h_{ij} - 1$ tests fail to meet the expectations, the probability that $h_{ij} - 2$ tests fail to meet the expectation, and so on. Therefore, the probability that e_{ij} has at least one error is the summation of the probabilities of each of the above scenarios; i.e., the failure probability of e_{ij} is given by

$$f_{ij} = \sum_{k=1}^{h_{ij}} C_{h_{ij}}^{k} q_{ij}^{k} p_{ij}^{h_{ij}-k}, \tag{11}$$

where

$$C_{h_{ij}}^{k} = \frac{h_{ij}!}{k!(h_{ij}-k)!}.$$

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

$$r_{ij} = 1 - f_{ij} = p_{ij}^{h_{ij}}. \tag{12}$$

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, and each edge has its reliability determined by the test results and is estimated by (12). 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. 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 serially connected or parallel-connected edges.

[1] m_{ij} can be a function of the failure metrics, such as the lines of code in e_{ij}, developer experience, or past performance, etc.
[2] The heuristic is that the more test scenarios the edge passes (the larger h_{ij} is), the lower failure probability the edge will have.

4.3 Reliability Estimation for Parallel Edge

First, for a block under node n_i composed of j_m parallel connected edges, the total number of executions of all parallel connected edges e_{ij} during the test stage is

$$h_i = \sum_{j=1}^{j_m} h_{ij} \tag{13}$$

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.,

$$T_i = \sum_{j=1}^{j_m} T_{ij} h_{ij} \,. \tag{14}$$

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

$$(p_{ij} + q_{ij})^{h_{ij}} = \sum_{k=0}^{h_{ij}} C_{h_{ij}}^{k} q_{ij}^{k} p_{ij}^{h_{ij}-k} = 1 \tag{15}$$

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 has a larger weight, and the summation of all of the weights is $\sum_{j=1}^{j_m} \frac{h_{ij}T_{ij}}{T_i} = 1.$ Therefore, the distribution of parallel edges should satisfy, considering (14) and (15), the following relationship:

$$\sum_{j=1}^{j_m} \frac{h_{ij}T_{ij}}{T_i} (p_{ij} + q_{ij})^{h_{ij}} = \sum_{j=1}^{j_m} \frac{h_{ij}T_{ij}}{T_i} = 1 \,. \tag{16}$$

Hence, immediately following (16), the reliability of the block composed of the parallel connected edges under node n_i is the event that every test has successfully passed. The probability of this event is given by

$$\begin{aligned} R_{i1} &= \sum_{j=1}^{j_m} \frac{h_{ij}T_{ij}}{T_i} p_{ij}^{h_{ij}} \\ &= \sum_{j=1}^{j_m} \frac{h_{ij}T_{ij}}{T_i} (1 - f_{ij}) \\ &= \sum_{j=1}^{j_m} \frac{h_{ij}T_{ij}}{T_i} r_{ij} \,. \end{aligned} \tag{17}$$

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,

$$H_{i1} = \sum_{j=1}^{j_m} h_{ij} \,. \tag{18}$$

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

$$T_{i1} = \frac{1}{H_{i1}} \sum_{j=1}^{j_m} T_{ij} h_{ij} \,. \tag{19}$$

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

4.4 Reliability Estimation for Serial Edge

For a block under node n_{i_1} composed of nodes $i_1,\ldots,i_s$ and serially connected edges (there are no parallel connected edges in all nodes $i_1,\ldots,i_s$), the total number of executions of all serially connected edges e_{ij} 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 serially 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.,

$$T_i = \sum_{i=i_1}^{i_s} T_{ij} h_{ij} \,. \tag{20}$$

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,

$$(p_{ij} + q_{ij})^{h_{ij}} = \sum_{k=0}^{h_{ij}} C_{h_{ij}}^{k} q_{ij}^{k} p_{ij}^{h_{ij}-k} = 1 \tag{21}$$

holds for all serially connected edges immediately under node n_{i_1}. For the serially connected edges, the reliability of the entire block is the product of the reliabilities of the individual edges. Considering (21), the following relationship immediately holds:

$$\prod_{i=i_1}^{i_s} (p_{ij} + q_{ij})^{h_{ij}} = 1 \tag{22}$$

Hence, the reliability of the block composed of serially connected edges under node n_{i_1} in the software is the event that every test has successfully passed. The probability of this event is given by

$$R_{i_1 j} = \prod_{i=i_1}^{i_s} p_{ij}^{h_{ij}} = \prod_{i=i_1}^{i_s} r_{ij} \,. \tag{23}$$

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 $i \in i_1,\ldots, i_s$, in the serial block, and

$$H_{i_1 j} = h_{ij} \,. \tag{24}$$

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

$$T_{i_1 j} = \sum_{i=i_1}^{i_s} T_{ij} \,. \tag{25}$$

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

4.5 Overall Reliability Estimation of the Software

The overall reliability of the software is estimated as follows. First, construct the flow network as discussed in Section 4.1. As testing proceeds, repeatedly use (9-12) 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 (16-19) to simplify parallel-connected edges into a single artificial edge and use equations (22-25) to simplify serially 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.

4.6 An Example

The pseudo c/c++ code example introduced in Section 4.1 is used to demonstrate how this software reliability estimation method works. The software partitioned as in Fig. 4 (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, $h_{11}=h_{41}=3$, $h_{21}=2$, and $h_{22}=h_{31}=h_{32}=1$. Assume $m_{ij}=2$ for all edges; thus, $p_{11}=p_{41}=0.999999$, $p_{31}=p_{32}=p_{22}=0.99$, and $p_{21}=0.9999$. 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, combining e_{31} and e_{32} gives $T_3=h_{31}T_{31}+h_{32}T_{32}=0.00002$, $\frac{h_{31}T_{31}}{T_3}=\frac{h_{32}T_{32}}{T_3}=\frac{1}{2}$; using (17) for the parallel edges e_{31} and e_{32}, the flow network is reduced to Fig. 4 (b) with $R_{31}=0.99$. Using (18) and (19), 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 (23) for the serial connection e_{21} and E_{31} to obtain the combined edge, the flow network is reduced to Fig. 4 (c) with $R_{21}=0.9999^2*0.99^2$. Using (24) and (25), 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. 4 (c), $T_2=H_{21}T_{21}+h_{22}T_{22}=0.00005$, $\frac{H_{21}T_{21}}{T_2}=\frac{4}{5}$, and $\frac{h_{22}T_{22}}{T_2}=\frac{1}{5}$, and using (14) for the parallel connection E_{21} and e_{22}, we reduce Fig. 4 (c) to Fig. 4 (d) with $R_{21}=\frac{0.9999^2*0.99^2*4}{5}+\frac{0.99*1}{5}$. Using (18) and (19), we obtain $H_{21}=\sum_{j=1}^{2}h_{2j}=3$ and $T_{21}=\frac{1}{3}\sum_{j=1}^{2}T_{2j}h_{2j}=0.00001*\frac{5}{3}$.
- Finally using (23) for serial connection e_1, E_{21}, and e_{41}, we have

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

4.7 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;

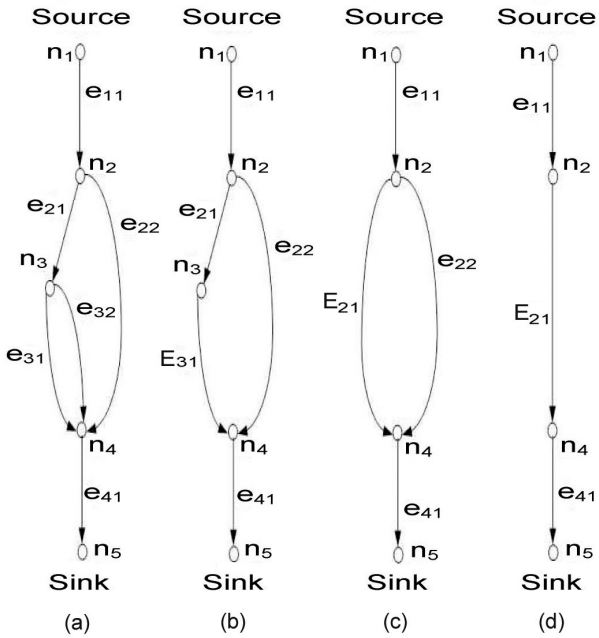

Fig. 4. Reliability Calculation Procedures

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

$$(1-R)\frac{t}{T}. \tag{26}$$

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, (26) continues to hold.

4.8 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 failure probability assignments, and reliability estimations, as it is tedious to count and record all $n_i \in N$, $e_{ij} \in E$, 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.

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.

It is proposed 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 h_{ij}, take average of T_{ij}, and accumulate T.
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 (9-12).
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} should be reset to one half, and q_{ij} and p_{ij} should be reset according to the new h_{ij}, after which the test will continue and all edge reliability estimation processes will be updated using (9-12) as before.
5. When all testing is complete, estimate the software reliability according to (13-25) and use the procedure presented in Section 4.5.
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.

5. FIRMWARE COMPONENT FAILURE MODEL

It is possible that some components will use firmware, such as a "smart sensor" or a board in a computer system, which has both hardware and software inside. Assume that the hardware has been tested and that the unreliability is obtained and given by (1); the software has been tested and the software reliability is evaluated using the method described in Section 4 and the unreliability is given by 1–*R*. Let A be the event of a software failure and B be the event of a hardware failure. Thus, the probability of a board failure is

$$p(A+B) = p(A) + p(B) - p(A \bullet B) \tag{27}$$

where

$$p(A \bullet B) = p(A \mid B)p(B) = p(B \mid A)p(A). \tag{28}$$

Because $\max(p(A \mid B), p(B \mid A))<1$, $p(A \bullet B)$ is smaller than $p=\min(p(A),p(B))$, it should be feasible to use the conservative "parts count" method described in [11] because the fault tree model evaluation is the order of magnitude of the failure rate. This gives

$$F(t) = (1-R)\frac{t}{T} + 1 - e^{-\lambda t}, \tag{29}$$

where the value of λ is obtained from (8).

6. ESTIMATE THE SYSTEM FAILURE RATE

6.1 Minimal Cut Set Unreliability

For single failures, it is clear that the minimal cut set unreliability is the component unreliability. For a minimal cut set with multiple failures, the minimal cut set unreliability

is simply determined by the multiplication of the unreliability of the components that are composed of the minimal cut set.

6.2 DI&C System Unreliability

Once we have determined the unreliability of all of the minimal cut sets, the unreliability of the entire DI&C system is the summation of the unreliability of all the minimal cut sets because the probability of two or more minimal cut sets occurring simultaneously is negligible [11, XI-19]. As indicated in Section 2, this method has the capability to handle common-cause failures.

7. CONCLUSIONS

In this paper, we proposed a systematic method to estimate the reliability of DI&C systems. A fault tree is used to model DI&C system unreliability, and common-cause failures can easily be treated in this model. Boolean algebra is used to derive the minimal cut sets. An exponential distribution is used to model hardware reliability. Bayesian estimation is used to estimate hardware failure rates. A binomial distribution and flow network are used to model software reliability, and testing is used to estimate the software failure rates. Using the hardware failure rates and the software failure rates, the firmware failure rates can be obtained by the parts-count method. These failure rates can be used to calculate the minimal cut set unreliability. Finally, the DI&C system unreliability can be obtained by the summation of all of the unreliability measures of the minimal cut sets.

ACKNOWLEDGMENTS

The authors would like to thank three anonymous reviews for their valuable comments, which lead to significant improvements to the presentation of the paper.

REFERENCES

[1] J. Dennis Lawrence, “Software Reliability and Safety in Nuclear Reactor Protection Systems”, US Nuclear Regulatory Commission, NUREG/CR-6001, 1993.
[2] David Lorge Parnas, G. J. K. Asmis, and Jan Madey, “Assessment of safety-critical software in nuclear power plants,” Nuclear Safety, Vol. 32, No. 2 pp.189-198 (1991).
[3] N. G. Leveson, P. R. Harvey, “Analyzing software safety,” IEEE Trans. On Software Engineering, Vol. 9, pp. 569-579, (1983).
[4] 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, pp71-117, 1996.
[5] S. Kuo, C. Huang, and M. Lyu, “Framework for modeling software reliability, using various testing-efforts and fault-detection rate,” IEEE Transactions on Reliability, Vol. 50, pp.310-320, 2001.
[6] H. Okamura, M. Ando, and T. Dohi, “A generalized gamma software reliability model,” Systems and Computers in Japan, Vol. 38, pp81-90, 2007.
[7] 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 Transactions on Reliability, Vol. 56, No. 3 pp. 411-421, (2007).
[8] C. Huang and C. Lin, “*Software Reliability Analysis by Considering Fault Dependency and Debugging Time Lag*,” IEEE Transactions on Reliability, Vol. 55, No. 2 pp. 436-450, (2006).
[9] IEC standard, IEC 61508 (all parts): Functional safety of electrical/electronic/programmable electronic safety-related systems, 2008.
[10] H. A. Watson, “Launch control safety study,” Bell Telephone Labs, Murray Hill, NJ USA, 1961.
[11] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, “Fault tree handbook,” US Nuclear Regulatory Commission, NUREG-0492, 1981.
[12] T. L. Chu, G. Martine-Guridi, M. Yue, and P. Samanta, “Traditional probabilistic risk assessment methods for digital system,” US Nuclear Regulatory Commission, NUREG/CR-6962, 2008.
[13] S. C. Bhatt and R. C. Wachowiak “ESBWR certification probabilistic risk assessment,” GE-Hitachi Nuclear Energy, NEDO-33201, Revision 2, 2007.
[14] A. E. Green and A. J. Bounme, *Reliability technology*, Wiley-Interscience, London, (1972).
[15] U.S. Nuclear Regulatory Commission, “Reactor safety study-an assessment of accident risks in U.S. commercial nuclear power plant”, NUREG-75/014, October, 1975.
[16] Department of Defense, “Reliability prediction of electronic equipment, Notice 2,” MIL-HDBK-217F, 1995.
[17] S. J. Press, *Bayesian statistics: principles, models, and applications*, John Wiley & Sons, New York, (1989).
[18] A. Elfessi and D. M. Eineke, “A Bayesian look at classical estimation: the exponential distribution,” *Journal of Statistics Education*, 2001, [online] 9(1). http://www.amstat.org/publications/jse/v9n1/elfessi.html
[19] J. H. Bickel, “Risk Implications of Digital Reactor Protection System Operating Experience,” Reliability Engineering & System Safety, Vol. 93, pp107-124 (2008).
[20] Y. Yang, “A flow network model for software reliability assessment,” Proceedings of 6th American nuclear society international topical meeting on nuclear plant instrumentation, control, and human-machine interface technologies (2009), Knoxville, April 5-9, 2009.
[21] T. L. Chu, M. Yue, G. Martinez-Gruidi, and J. Lehner, Review of quantitative software reliability methods, BNL-94074-2010, Brookhaven National Laboratory (2010).
[22] Y. Yang and R. Sydnor, “Multi-threads software reliability estimation based on test results and software structure,” Proceedings of 10th international probabilistic safety assessment and management conference (2010), Seattle, June 7-11, 2010.