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 reliability 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.

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...
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:

Using the symbols defined in Fig. 2, the fault tree model can be converted to Boolean equations, as follows:

\[
\begin{align*}
S &= M + N + DA \\
DA &= K + L + P \\
P &= I + J + AD \\
AD &= H + G + S1 \cdot S2 + S2 \cdot S3 + S1 \cdot S3 + S1 \cdot S2 \cdot S3 \\
S1 &= A + B + A + X \\
S2 &= C + D + C + X \\
S3 &= E + F + E + X
\end{align*}
\]

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

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

(2)

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

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

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 $\lambda$ 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 $\lambda$. 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

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

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

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

For the sake of algebraic convenience, a conjugate priori distribution of the failure rate $\lambda$ is suggested for $\lambda$, which results in

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

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

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

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

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

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 \( n \in N \) represent the logic branch or converging points; and edges \( e, j = 1, \ldots, j_n \) 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.

```c
Main(){ //node 1
  Data initialization; //edge e_11
  If condition A holds //node 2
  { 
    Process data; //edge e_21
    If data process success //node 3
    { 
      Save result; //edge e_31
    }
    Else if data process fail
    { 
      Issue a warning; //edge e_32
    }
  }
  Else if condition A does not hold
  { 
    Print “condition fail”; //edge e_22
  }
  Clean memories; //edge e_41 //node 4
}
//node 5
```

By applying the principles described above, the flow network model corresponding to the pseudo code is given in Fig. 3.


4.2 Reliability Estimation for a Single Edge

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

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

$$q_{b} = 10^{-m_{b}}.$$  \hfill (9a)

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

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 $e_{i}$ of the software multiple times and the test results meet the expectations, then the estimated software failure probability for $e_{i}$ should be reduced as more tests meet the expectations. Moreover, if edge $e_{i}$ is executed more times than edge $e_{j}$ in the test scenarios and if all of these tests involving $e_{i}$ meet the test expectations, the failure probability of edge $e_{i}$ should be smaller than the failure probability of edge $e_{j}$, i.e., $q_{i} > q_{j}$ or $p_{i} < p_{j}$. Therefore, if $e_{i}$ is executed exactly $h_{i}$ times and all tests meet the expectations during the software test stage, the failure probability of $e_{i}$ is defined as

$$q_{i} = 10^{-h_{i}m_{i}},$$  \hfill (9b)

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

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

$$p_{i} = 1 - q_{i}$$.  \hfill (10)

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

$$f_{y} = \sum_{k=1}^{h_{i}} C_{h_{i}}^{k} q_{i}^{k} p_{i}^{h_{i}-k},$$  \hfill (11)

where

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

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

$$r_{y} = 1 - f_{y} = p_{i}^{h_{i}},$$  \hfill (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 $m_{i}$ can be a function of the failure metrics, such as the lines of code in $e_{i}$, developer experience, or past performance, etc.

2 The heuristic is that the more test scenarios the edge passes (the larger $h_{i}$ 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

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

(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.,

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

(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,

$$(p_y + q_y)^{h_{ij}} = \sum_{k=0}^{h_{ij}} C_{h_{ij}}^k q_y^{h_{ij}-k} = 1$$

holds for every parallel connected edge immediately under node $n_i$. As every edge in the parallel structure has its own binomial distribution $(p_y + q_y)^{h_{ij}}$ and its own execution time $h_i T_{ij}$, 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 $\frac{h_i T_{ij}}{T_i}$; i.e., an edge that has longer execution time has a larger weight, and the summation of all of the weights is $\sum_{j=1}^{j_m} \frac{h_i T_{ij}}{T_i} = 1$. Therefore, the distribution of parallel edges should satisfy, considering (14) and (15), the following relationship:

$$\sum_{j=1}^{j_m} \frac{h_i T_{ij}}{T_i} (p_y + q_y)^{h_{ij}} = \sum_{j=1}^{j_m} \frac{h_i T_{ij}}{T_i} = 1.$$  

(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

$$R_{ni} = \sum_{j=1}^{j_m} \frac{h_i T_{ij}}{T_i} p_y^{h_{ij}}$$

$$= \sum_{j=1}^{j_m} \frac{h_i T_{ij}}{T_i} (1 - f_y)$$

$$= \sum_{j=1}^{j_m} \frac{h_i T_{ij}}{T_i} r_y.$$  

(17)

The number of executions of the combined artificial edge used in the next model reduction step is

$$H_{ni} = \sum_{j=1}^{j_m} h_{ij}.$$  

(18)

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 $n_i$ 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.,

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

(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,

$$(p_y + q_y)^{h_{ij}} = \sum_{k=0}^{h_{ij}} C_{h_{ij}}^k q_y^{h_{ij}-k} = 1$$

holds for all serially connected edges immediately under node $n_i$. 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:

$$\prod_{i=1}^{j_m} (p_y + q_y)^{h_{ij}} = 1.$$  

(22)

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

$$R_{ki} = \prod_{j=1}^{j_m} p_y^{h_{ij}} = \prod_{j=1}^{j_m} r_y.$$  

(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

$$H_{ki} = h_{ij}.$$  

(24)

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

$$T_{ki} = \sum_{j=1}^{j_m} T_{ij}.$$  

(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 $e_{10}e_{20}e_{31}e_{41}$, the second test path is $e_{10}e_{20}e_{32}e_{41}$, and the third test path is $e_{10}e_{20}e_{41}$. Assume further that the total test time is $T=0.00011$ hours and $T_{ij}=0.00001$ hours for every edge. Therefore, $h_{11}=h_{41}=3$, $h_{31}=2$, and $h_{32}=h_{51}=h_{52}=1$. Assume $m_{ij}=2$ for all edges; thus, $p_{11}=p_{41}=0.999999$, $p_{31}=p_{32}=p_{52}=0.99$, and $p_{12}=0.9999$. 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 $T_2=h_{31}T_{31}+h_{32}T_{32}=0.00002$, $h_{31}T_{31}=h_{32}T_{32}=1/2$, 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 $H_{31} = \sum h_{ji} = 2$, and $T_{31} = \sum i T_{ij} h_{ij} = 0.00001$.

- 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 $R_{21}=0.9999992 * 0.992$. Using (24) and (25), we have $T_{21} = \sum T_{ji} = 0.000002$, and $H_{21}=h_{21}H_{31}=2$.

- Considering the parallel connection in Fig. 4 (c), $T_2=H_{21}T_{21}+h_{22}T_{22}=0.000005$, $h_{21}T_{21} = 4/5$ and $h_{22}T_{22} = 1/5$ and using (14) for the parallel connection $E_{21}$ and $e_{22}$, we reduce Fig. 4 (c) to Fig. 4 (d) with $R_{21} = \frac{0.9999992 * 0.992 * 4 + 0.99 * 1}{5}$.

Using (18) and (19), we obtain $H_{21} = \sum h_{ji} = 3$ and $T_{21} = \sum i T_{ij} h_{ij} = 0.00001 \times \frac{5}{3}$.

- Finally using (23) for serial connection $e_{11}$, $E_{21}$, and $e_{41}$, we have $R = 0.9999999 \times \left( \frac{0.9999992 * 0.992 * 4 + 0.99 * 1}{5} \right) \times 0.9999999 = 0.981917$.

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 $T$;
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 \(n_i \in N\),
\(e_{ij} \in E\), \(T_{ni}\), \(T_i\), and \(h_{ij}\) values manually; to update all \(q_{ij}\) and
\(p_i\) 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 \(T_{ni}\), \(T_i\), 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 \(T_{ni}\) and accumulate \(T_i\).
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 \(p_i\) 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 \(p_i\) 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
\[
p(A \bullet B) = p(A | B)p(B) = p(B | A)p(A). \tag{28}
\]

Because \(\max(p(A | B), p(B | A)) \leq 1\), \(p(A \bullet B)\) 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 \(\lambda\) 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 reviewers for their valuable comments, which lead to significant improvements to the presentation of the paper.

REFERENCES