Fault-Tolerant Programs and Their Reliability

Fevzi Belli
University of Paderborn, Paderborn

Piotr Jedrzejovicz
Merchant Marine Academy, Gdynia

Key Words — Software reliability, Fault-tree analysis, Software-fault tolerance, n-version programming, Recovery-block scheme, Software fault, Program construction

Reader Aids —
Purpose: Survey & widen the state of the art
Special math needed for explanations: Elementary probability
Special math needed to use results: Same
Results useful to: Software & reliability engineers, system analysts

Summary & Conclusions — The paper reviews and extends available techniques for achieving fault-tolerant programs. The representation of the techniques is uniform and is illustrated by simple examples. For each technique a fault tree has been developed to derive failure probability from the probabilities of the basic fault events. This enables the subsequent analysis of program-failure causes and the reliability modeling of computer programs. Numerical examples support the comparison of the reviewed techniques.

The models can be used to evaluate numerical values of program reliability in a relatively simple way. The models deal with program reliability for a single run which seems more practical and straightforward than dealing with distributions as for hardware systems. Evaluations obtained by using our models correspond to those used in the literature. Yet our procedures are computationally simpler.

1. INTRODUCTION

Increasing the redundancy within the software system is a common way to achieve fault tolerance. Redundant components, however, require additional resources, e.g., additional costs in terms of programming effort, hardware requirements, and time needed for design and test. During software-system operation, redundancy causes longer processing times. Therefore the redundancy level needed to achieve fault tolerance must be carefully determined and, if possible, optimized. This is still hardly possible due to the lack of relevant reliability models.

Only a few techniques are known to be useful when constructing fault-tolerant software systems. Two of the most widely discussed in the literature are recovery block scheme and n-version programming. We discuss both, suggest some extensions, and propose an approach towards reliability modeling based on fault-tree analysis.

Acronyms

RBS recovery block scheme
NVP n-version programming

Nomenclature

Computer program — A collection of modules to perform the overall function.
Module — A sequence of computer instructions that performs a well-defined function as an element of a computer program. A module can be compiled independently of other modules and of the main program.
Segment — A submodule that performs a subfunction of the module.
(The terms module, segment, and path are often used interchangeably.)

2. RECOVERY BLOCK SCHEMES

2.1 Simple RBS

The following simple example illustrates the well known concept of the Simple RBS [1, 4]. In figure 1, the program compare compares two given integers nb1 and nb2 and is written in a Pascal-like language. This program consists of the procedures comp1 and accept_test. The comp1 and comp2 realize the same function in different code, and accept__test represents an acceptance test whose role is to accept/reject the computation result. The test is absolute in the sense that it results precisely in true or false.

If comp1 is not accepted then comp2 is activated. If it also fails then an error (not specified here) is output. Figure 1 is the program listing.

Assumptions

• Each program consists of n modules.
• Each module consists of a sequence of instructions (functional segment).
• There exists a testing and recovery segment checking against the result of computations at the end of module #1. If the acceptance test detects an erroneous output then the input state is recovered and module #2 activated. The procedure is repeated until acceptance, or the acceptance test judges the last module as incorrect (regardless of its actual correctness). Figure 2 is the fault tree for this scheme. The testing and recovery segment can, itself, be viewed as a module.

Notation

\( e_i \quad \text{probability that the functional module} \ i \ \text{has failed} \)
\( t_i \quad \text{probability that the testing segment cannot perform recovery upon functional module-failure. This includes} \)
PROGRAM compare;

PROCEDURE comp1;
...
BEGIN (* comp1 *)
  IF nb1 > nb2
    THEN max := nb1
    ELSE max := nb2;
  END; (* comp1 *)
...

PROCEDURE comp2;
...
BEGIN (* comp2 *)
  booly := nb1 > nb2;
  CASE booly OF
    TRUE: max := nb1;
    FALSE: max := nb2
  END; (* case *)
...
END; (* comp2 *)

PROCEDURE accept_test;
...
BEGIN (* accept_test *)
  oracle := (max >= nb1)
  AND
    (max >= nb2);
...
END; (* accept_test *)

...
BEGIN (* main: compare *)
...
oracle, booly := FALSE;
...
comp1;
  accept_test;
END;
...
BEGIN (* comp2 *)
  accept_test;
END;
...
END; (* compare *)

Figure 1. Example of the Simple RBS

Figure 2. Fault Tree for the Simple RBS.

\[
F_1 = \prod_{i=1}^{n} e_i
\]  
\[
F_2 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (-1)^{n-2} \cap_{i=1}^{n} x_i
\]

\[
x_i = t_1 \prod_{k=1}^{i} (e_1 + t_2 - e_{f_2})
\]

2.2 Generalized RBS

The Simple RBS can be generalized by adding its own test module to each functional module. This is demonstrated by the example program in figure 3 (which extends the program in figure 1) and the fault tree in figure 4.

Additional Assumptions (to Simple RBS)

- A test module is attached to each regular module (excluding module \(n\)). The regular module includes instructions which perform the module functions. The test module performs an absolute test of the regular module. The role of the test module is to: 1) test whether the instructions of the regular module have been realized error free, and 2) execute a recovery function if an error has been detected. The test module can fail in 2 modes; i.e., the regular module is evaluated as 1) erroneous but errors have not occurred, or 2) correct but errors have occurred. If the error is detected by the test module for regular module \(i(i=1,\ldots,n-1)\) then the initial state of module \(i\) is recovered, and regular module \(i+1\) is activated and run.

- Regular module \(n\) has no test module.

The failure probability of the program is:

\[
F = F_1 + F_2 - F_1 F_2
\]
PROGRAM compare;
  ...
PROCEDURE comp1;
  ...
BEGIN (* comp1 *)
  IF nb1 < nb2
    THEN max := nb1
    ELSE max := nb2;
  ...
END; (* comp1 *)

PROCEDURE comp2;
  ...
BEGIN (* comp2 *)
  booly := nb1 > nb2;
  CASE booly OF
    TRUE: max := nb1
    FALSE: max := nb2
  END; (* case *)
  ...
END; (* comp2 *)

PROCEDURE accept_test1;
  ...
BEGIN (* accept_test1 *)
  oracle := \( max \geq nb1 \) AND \( max \geq nb2 \);
  ...
END; (* accept_test1 *)

PROCEDURE accept_test2;
  ...
BEGIN (* accept_test2 *)
  oracle := \( nb2 \leq max \) AND \( nb1 \leq max \);
  ...
END; (* accept_test2 *)

BEGIN (* compare *)
  booly := FALSE;
  ...
  comp1;
  accept_test1;
  IF NOT oracle THEN BEGIN
    comp2;
    accept_test2;
    IF NOT oracle THEN <error>
  END; (* if *)
END. (* compare *)

Figure 3. Example of the Generalized RBS.

Figure 4. Fault Tree for the Generalized RBS.

The failure probability of the program is:

\[
F = F_1 + F_2 - F_1 F_2
\]

(5)

\[
F_1 = e_n \prod_{i=1}^{n-1} (e_i + r_{2i} - e_f r_{2i})
\]

(6)

\[
F_2 = \sum_{i=1}^{n-1} x_i - \sum_{i=1}^{n-1} \sum_{j<i} (x_i \cap x_j) + \ldots + (-1)^{n-2} \bigcap_{i=1}^{n-1} x_i,
\]

(7)

\[
x_i = t_{1i} \prod_{k=1}^{i} (e_k + r_{2k} - e_f r_{2k}); \quad i = 1, \ldots, n-1,
\]

(8)

2.3 Concurrent RBS

Figures 5 & 6 depict the example program and the fault-tree diagram of this scheme.

Notation

- \( e_i \): probability that the regular module at path \( i (i = 1, \ldots, n) \)
  is failed
- \( r_{1i} \): probability that the test module at path \( i (i = 1, \ldots, n-1) \)
  cannot perform recovery upon failure of the regular module. This includes the
  failure mode in which test

module erroneously judges an incorrect result as correct and therefore the recovery
is not carried out probability that the test module at path \( i (i = 1, \ldots, n-1) \)
erroneously judges correct results as errors.
PROGRAM compare;
...
PROCEDURE comp_accept1;
...
BEGIN (* comp_accept1 *)
(* comp1 *)
    IF nb1 < nb2
    THEN max := nb1
    ELSE max := nb2;
...
(* accept_test1 *)
    oracle := (max >= nb1)
    AND
    (max >= nb2);
END: (* comp_accept1 *)

PROCEDURE comp_accept2;
...
BEGIN (* comp_accept2 *)
(* comp2 *)
    booly := nb1 > nb2;
    CASE booly OF
    TRUE: max := nb1;
    FALSE: max := nb2
END: (* case *)
(* accept_test2 *)
    oracle := (nb2 <= max)
    AND
    (nb1 <= max);
END: (* comp_accept2 *)
...
BEGIN (* main: compare *)
    oracle, booly := FALSE;
    ACTIVATE comp_accept1, comp_accept2;
    IF NOT oracle THEN <error>;
END: (* main *)

Figure 5. Example of the Concurrent RBS.

The failure probability of the program is:

\[
F = F_0 + F_1 - F_0 F_1
\]  

(9)

\[
F_0 \equiv \prod_{i=1}^{n} (e_i + g_i - e_i g_i),
\]  

(10)

\[
F_1 = \sum_{i=1}^{n} F_i - \sum_{j=1}^{n} (F_i \cap F_j) + \ldots + (-1)^{n-1} \bigcap_{i=1}^{n} F_i,
\]  

(11)

\[
F_i = h_i \prod_{k=1}^{i} e_k; \quad i = 1, \ldots, n-1
\]  

(12)

3. N-VERSION PROGRAMMING

This approach [3] can be demonstrated by the simple example in figure 7, and its fault tree in figure 8. As in previous examples, modules comp1 and comp2 have identical functions. The procedure decide is a simple decision mechanism that checks the validity of the results of comp1 and comp2. If the result of either is valid the next program is activated; else an error message is output. The validity of the result is indicated by the global parameter oracle.

Assumptions

• Each program consists of n "parallel" modules that can run concurrently.
• Each module is a different sequence of instructions.
• The decision algorithm (in this case a simple voter) selects the first set of results that pass the test. If the set of similar errors outnumbers the set of good (similar) results at a decision point then the decision is erroneous.
• A decision algorithm may itself fail by not being able to select the correct result out of two or three proper results.

Notation

\( e_i \) probability that the functional module at path \( i(i=1,\ldots,n) \) is failed
\( d \) probability that the decision algorithm cannot select the result from at least 2 correct results

The failure probability of the program is:

\[
F = \sum_{i=1}^{n+1} f_i - \sum_{i=1}^{n+1} \sum_{j>i} f_i \cap f_j + \ldots + (-1)^{n} \bigcap_{i=1}^{n+1} f_i
\]  

(13)

\[
f_k \equiv e_k^{n-k+1} \prod_{i=1}^{n} e_i; k = 1, \ldots, n,
\]  

(14)

\[
f_{n+1} = d.
\]  

(15)
4. COMBINED FAULT-TOLERANT TECHNIQUES

4.1 Combined RBS & NVP

The ideas behind Plain RBS and NVP can be combined. We illustrate this combination by extending the examples in sections 2 & 3. The example program is in figure 9; its fault tree is in figure 10.

This structure should be used when the testing modules within the traditional recovery-block schemes are unreliable, e.g., because of their over-simplicity or due to difficulties in evaluating the functional module performance.

Assumptions

- A program consists of \( n \) paths and a decision algorithm.
- Each path consists of 2 modules: Functional and test.
- All paths can run concurrently.
- Computation results at each path are immediately evaluated by the respective test module. Erroneous results are discarded from further consideration, without executing recovery.
- The remaining set of possibly correct results is subjected to a decision (majority rule) algorithm.

\[
F = \sum_{i=1}^{n} f_i - \sum_{i=1}^{n} \sum_{j>i}^{n+1} (f_i \cap f_j) + \ldots + (-1)^{n+1} f_i
\]

\[
f_k = \prod_{i=1}^{k} \left( e_i + g_i + h_i - e_i g_i - e_i h_i - g_i h_i + e_i g_i h_i \right)
\]

\[
f_{n+1} = d.
\]

4.2 Consensus RBS

This scheme, proposed by Scott, Gault, McAllister [5], requires the \( n \)-version of a program, a testing segment performing the acceptance check, and a decision algorithm with a voting procedure. The test procedure is performed as a sequence of stages.

Assumptions

- Each version of a module representing the functional part of a program is mutually independent of all other versions.
- All paths (functional modules) can run concurrently.
- All paths are ranked.
- Upon invocation of the Consensus RBS, all paths execute and submit their outputs to a decision algorithm. If at least 2 results agree on one output, that output is designated as correct. Otherwise the next stage is entered.

![Figure 7. Example of NVP.](image-url)

![Figure 8. Fault Tree NVP.](image-url)
PROGRAM compare;
      ...
PROCEDURE comp1;
      ...
BEGIN (* comp1 *)
IF nb1 > nb2
THEN max1 := nb1
ELSE max1 := nb2;
      ...
END; (* comp1 *)
      ...
PROCEDURE comp2;
      ...
BEGIN (* comp2 *)
booly := nb1 > nb2;
CASE booly OF
  TRUE: max2 := nb1;
  FALSE: max2 := nb2;
END; (* case *)
      ...
END; (* comp2 *)
PROCEDURE comp3;
      ...
BEGIN (* comp3 *)
max3 := nb1;
IF max3 < nb2
THEN max3 := nb2;
      ...
END; (* comp3 *)
      ...
PROCEDURE accept_test1;
BEGIN (* accept_test1 *)
    oracle := (max1 >= nb1)
AND
      (max1 >= nb2);
      ...
END; (* accept_test1 *)
PROCEDURE accept_test2;
BEGIN (* accept_test2 *)
    oracle := (nb2 <= max2)
AND
      (nb2 <= max2);
      ...
END; (* accept_test2 *)
PROCEDURE accept_test3;
BEGIN (* accept_test3 *)
    oracle := (nb1 <= max3)
AND
      (max3 >= nb2);
      ...
END; (* accept_test3 *)
PROCEDURE decide;
      ...
BEGIN (* decide *)
      ...
    oracle := (max1 = max2) OR (max1 = max3) OR (max2 = max3);
      ...
END; (* decide *)
      ...
BEGIN (* main: compare *)
      ...
    oracle, booly := FALSE;
      ...
ACTIVATE comp1, accept_test1, comp2, accept_test2, comp3, accept_test3, decide;
      IF NOT oracle THEN <error>;
      ...
END; (* compare *)

Figure 9. Example of Combined RBS & NVP.

Figure 10. Fault Tree for the Combined RBS & NVP.

- At this stage the "best" path is examined by the acceptance test. If that output of that stage is accepted then it is treated as correct, else the next best version output is subject to testing. This process continues until an acceptable output is found, or the n outputs are exhausted.

An example program and the fault tree are shown in figures 11 and 12.

Notation

\( e_i \)  probability that the functional module at path \( i(i=1,...,n) \) is failed
\( d \)  probability that the decision algorithm cannot select the result out of at least 2 correct results
\( t \)  probability that every correct result is rejected

The failure probability of the program is:

\[
F = F_0 \cdot F_1
\]

\[
F_0 = \sum_{i=1}^{n+1} \sum_{j=1}^{n} (f_i \cap f_j) + \ldots + (-1)^{n+1} \bigcap_{i=1}^{n+1} f_i
\]

\[
f_k = e_k \prod_{i=1}^{k} e_i, \quad k = 1,...,n,
\]

\[
f_{n+1} = d,
\]

\[
F_1 = \prod_{i=1}^{n} (e_i + t_i - e_i t_i).
\]

5. EXAMPLES & COMPARISONS

5.1 Example 1

The example in figure 13 compares the reliability of the fault-tolerant models treated in sections 2-4. The components of each model are similar.
Assumptions

- Each program has 3 functional modules, each with reliability = 0.95.
- Each test module or decision algorithms has reliability = 0.99.

```
PROGRAM compare;
...
PROCEDURE comp1;
...
BEGIN (* comp1 *)
IF nb1 > nb2
THEN max1 := nb1
ELSE max1 := nb2;
...
END; (* comp1 *)

PROCEDURE comp2;
...
BEGIN (* comp2 *)
booly := nb1 > nb2;
CASE booly OF
  TRUE: max2 := nb1;
  FALSE: max2 := nb2;
END; (* case *)
...
END; (* comp2 *)

PROCEDURE comp3;
...
BEGIN (* comp3 *)
max3 := nb1;
IF max3 < nb2
THEN max3 := nb2;
...
END; (* comp3 *)

PROCEDURE accept_test;
BEGIN (* accept_test *)
oracle := (max1 >= nb1) AND (max1 >= nb2);
...
END; (* accept_test *)

PROCEDURE decide;
...
BEGIN (* decide *)
oracle := (max1 = max2) OR (max1 = max3) OR (max2 = max3);
...
END; (* decide *)

BEGIN (* main: compare *)
...
oracle, booly := FALSE;
ACTIVATE comp1, comp2, comp3;
declare;
IF NOT oracle THEN <ENSURE accept_test
  BY comp1
ELSE comp2
ELSE comp3
ELSE <error>>;
...
END; (* compare *)
```

Figure 11. Program for Consensus RBS.

![Fault Tree for the Consensus RBS](image)

Figure 12. Fault Tree for the Consensus RBS.

Although, because of the different structures and various functions of the components, the comparison of models requires more than merely analyzing program reliability, the results clearly show the advantage of the Consensus RBS for this example.

5.2 Example 2

Consensus RBS assures good behavior when the functional modules are inferior. Figure 14 shows the dependency between program reliability and the functional reliabilities for the best models, viz, Concurrent RBS and Generalized RBS. All the remaining values of the basic fault events are constant and equal to those in figure 13 for example 1.

5.3 Example 3

The example in figure 15 compares the results obtained by Scott, Gault, McAllister [5] for Consensus RBS with those from our models. To make comparisons possible, we made the following assumptions:

Assumptions

- Reliabilities of the decision algorithms for n-version programming and Consensus RBS are always 1
- Testing segments for Simple RBS and Consensus RBS have only one type of failure.

5.4 Importance of an Element

Figure 16 summarizes the importance of elements or groups of elements used to construct fault-tolerant programs. The importance of an element from the system viewpoint is:
### Figure 13. Probabilities of Basic Faults and the Program Reliabilities.

<table>
<thead>
<tr>
<th>Model</th>
<th>$e_i$</th>
<th>$t_i$</th>
<th>$t_j$</th>
<th>$t_{1i}$</th>
<th>$t_{2i}$</th>
<th>$g_i$</th>
<th>$h_i$</th>
<th>$d$</th>
<th>$t$</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBS</td>
<td>.05</td>
<td>.01</td>
<td>.01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>.99919</td>
</tr>
<tr>
<td>CRBS</td>
<td>.05</td>
<td>-</td>
<td>-</td>
<td>.01</td>
<td>.01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>.99919</td>
</tr>
<tr>
<td>CnctRBS</td>
<td>.05</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>.01</td>
<td>.01</td>
<td>-</td>
<td>-</td>
<td>.99926</td>
</tr>
<tr>
<td>NVP</td>
<td>.05</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>.01</td>
<td>-</td>
<td>.98259</td>
</tr>
<tr>
<td>RBS-NVP</td>
<td>.05</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>.01</td>
<td>.01</td>
<td>.01</td>
<td>-</td>
<td>-</td>
<td>.97596</td>
</tr>
<tr>
<td>CRBS</td>
<td>.05</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>.01</td>
<td>.01</td>
<td>.99999</td>
</tr>
</tbody>
</table>

$[R = \text{reliability of a functional module}]$

### Figure 14. Reliability of the Generalized RBS and Concurrent RBS.

<table>
<thead>
<tr>
<th>Model</th>
<th>$e_1$</th>
<th>$e_2$</th>
<th>$e_3$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t$</th>
<th>$d$</th>
<th>$R$ as in [5]</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVP</td>
<td>.02</td>
<td>.07</td>
<td>.09</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>.9908</td>
<td>.9905</td>
</tr>
<tr>
<td>RBS</td>
<td>.02</td>
<td>.20</td>
<td>.30</td>
<td>.07</td>
<td>.07</td>
<td>-</td>
<td>-</td>
<td>.9900</td>
<td>.9904</td>
</tr>
<tr>
<td>CRBS</td>
<td>.02</td>
<td>.40</td>
<td>.40</td>
<td>-</td>
<td>-</td>
<td>.07</td>
<td>0</td>
<td>.9905</td>
<td>.9918</td>
</tr>
</tbody>
</table>

### Figure 15. Reliabilities for Alternative Models.

### Figure 16. Importance of Components in Example 3.

<table>
<thead>
<tr>
<th>Item</th>
<th>RBS</th>
<th>GRBS</th>
<th>CnctRBS</th>
<th>NVP</th>
<th>RBS-NVP</th>
<th>CRBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>.0134</td>
<td>.0134</td>
<td>.0140</td>
<td>.0963</td>
<td>.1418</td>
<td>.0003</td>
</tr>
<tr>
<td>$R_1, ..., R_{N-1}$</td>
<td>.0088</td>
<td>.0868</td>
<td>.0787</td>
<td>.9999</td>
<td>.9868</td>
<td>.0595</td>
</tr>
<tr>
<td>$t_1$</td>
<td>.0628</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$t_2$</td>
<td>.0683</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>all $t_1$</td>
<td>-</td>
<td>.0683</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>all $t_2$</td>
<td>-</td>
<td>.0628</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$g_1$</td>
<td>.0011</td>
<td>.0011</td>
<td>.0005</td>
<td>.1231</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$h_1$</td>
<td>.9923</td>
<td>.9838</td>
<td>.0003</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Figure 17. Importance of the Redundant Modules for Figure 16.

<table>
<thead>
<tr>
<th>Item</th>
<th>RBS</th>
<th>GRBS</th>
<th>CnctRBS</th>
<th>NVP</th>
<th>RBS-NVP</th>
<th>CRBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_0$</td>
<td>.0022</td>
<td>.0022</td>
<td>.0526</td>
<td>.0891</td>
<td>.1173</td>
<td>.0004</td>
</tr>
</tbody>
</table>
\[ I_R = \frac{\partial R}{\partial R_i} = R(R_i = 1) - R(R_i = 0) \]  \hspace{1cm} (24)

**Notation**

- \( R \) system reliability
- \( R_i \) reliability of component \( i \)

Figure 17 shows the importance of the redundant functional modules for the models in figure 16. Importance, here, is defined as:

\[ I_n = R(n) - R(n-1) \]  \hspace{1cm} (25)

**Notation**

- \( n \) number of functional modules in the program

By increasing program redundancy the biggest reliability growth is anticipated for the Combined RBS & NVP. Indeed, adding a 4-th additional functional module to the example program, changes the system reliability from the 0.9759 in figure 13, to 0.9887 — over a factor of 2 improvement in unreliability.

6. FAULTS AND THEIR INTERACTIONS

Faults and their interactions in various fault-tolerant programs are summarized as follows.

**Simple RBS fails if:**
- Each functional module fails or is not accepted; OR
- The first functional module, or a set of the consecutive modules including the first one, fails or is not accepted AND the test module cannot perform recovery.

**Generalized RBS fails if:**
- Each functional module fails or is not accepted; OR
- The first functional module, or a set of the consecutive modules including the first, fails or is not accepted AND the testing segment cannot perform recovery.

**Concurrent RBS fails if:**
- Each functional module fails or is not accepted; OR
- The first functional module, or a set of the consecutive modules including the first, fails and is erroneously accepted.

**N-Version Programming fails if:**
- For some \((n-1)\)-tuple of functional modules, all fail; OR
- The decision algorithm is not able to select the result out of at least 2 correct results.

Combined RBS & NVP fails if:
- For some \((n-1)\)-tuple of functional modules, all fail or are not accepted; OR
- The decision algorithm is not able to select the result out of at least 2 correct results.

Consensus RBS fails if:
- For some \((n-1)\)-tuple of functional modules all fail, OR the decision algorithm is not able to select the result out of at least 2 correct results; AND
- Each functional module fails or is not accepted.

REFERENCES


AUTHORS

Prof. Dr. Ing. habil. Fevzi Belli; Dept. of Electrical & Electronics Engineering; The University of Paderborn; Pohlweg 47/49; D-4790 Paderborn; Fed. Rep. GERMANY.

F. Belli, born 1948, is a professor of Computer Science at the University of Paderborn and is a faculty member of the University of Maryland, European Division. His research interests are in software fault tolerance, software reliability, programming techniques, and knowledge-based systems. His degrees (MS, PhD, Habilitation) are from Berlin Technical University in Electricity and Computer Science. He is a member of IEEE Computer Society (since 1983). ACM, German Association for Informatics and its working groups “Software Reliability” and “Fault-Tolerant Computing”, German Association for Engineering, etc.

Prof. Dr. habil. Piotr Jedrzejowicz; Dept. of Systems Analysis & Informatics; Merchant Marine Academy; ul. Czernowich Kosynierów; 83 Gdynia; POLAND.

P. Jedrzejowicz, born 1942, is a professor of Computer Science and Head of the Merchant Marine Academy in Gdynia. He served as head of the Department of Systems Analysis & Informatics of the Academy. His degrees (MA, PhD, Habilitation) are from the University of Gdansk in Operations Research and Maritime Economics. His research interests are in reliability optimization, modeling and simulation of complex systems, and computer-supported decision systems.