Detecting Common Mode Failures in N-Version Software Using Weakest Precondition Analysis *

Gwang Sik Yoon, Sung Deok Cha, and Yong Rae Kwon
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon 305-701, Korea
{gsyoon,cha,yrkwon}@salmosa.kaist.ac.kr

Chan Hyoung Yoo
ATM M&A Software Section
Electronics and Telecommunications Research Institute
161 Kajong-dong, Yusong-gu, Taejon 305-350, Korea
chyoo@nice.etri.re.kr

Abstract

An underlying assumption for N-version programming technique is that independently developed versions would fail in a statistically independent manner. However, empirical studies have demonstrated that common mode failures can occur even for independently developed versions, and that common mode failures degrade system reliability.

In this paper, we demonstrate that the weakest precondition analysis is effective in determining input spaces leading to common mode failures. We applied the weakest precondition to the Launch Interceptor Programs which were used in several other experiments related to the N-version programming technique. We detected 13 out of 18 fault pairs which have been known to cause common mode failure. These faults were due to logical flaws in program design. Although the weakest precondition analysis may be labor-intensive since they are applied manually, our results convincingly demonstrate that it is effective for identifying input spaces causing common mode failures and further improving the reliability of N-version software.

1. Introduction

The N-version programming(NVP) is a software fault tolerance technique being widely applied to safety-critical industrial systems such as the Airbus flight control system, railway interlocking system and train control system [3]. An N-version system(NVS) is composed of more than one functionally equivalent programs. It is based on the assumption that independent programming efforts will greatly reduce the probability of producing identical software faults in two or more versions and that each version would fail in a statistically independent manner [1].

Common mode failures(CMF) occur when more than one version fail simultaneously on the same input producing identical incorrect output. Reliability of an NVS is reduced substantially when CMF exists [7]. Several approaches have been suggested to develop versions that would fail independently [10, 2, 11]. However their effectiveness in preventing CMF during the development phase has not been verified, and according to independent experiments, common mode failures are known to exist regardless of approaches taken [8, 12].

In this paper, we propose to use the weakest precondition analysis to detect inputs causing CMFs. The weakest precondition analysis is a well known technique used to prove the correctness of programs [6, 9]. It is a formal backward analysis method and the effectiveness of analysis results does not depend upon the domain knowledge of the analyst. The overall process of our approach can be divided into a number of phases. First, a postcondition representing program failures is specified in terms of output variables. Then each version is analyzed backward, and the weakest preconditions are derived. The derived weakest preconditions are validated with respect to the specification and conditions for CMF are identified. Finally, the identified conditions causing CMF can be used in testing and debugging. Feasibility of the proposed method was verified by conducting an experiment with realistic example programs. In our ex-

*This research is supported in part by ETRI(Electronics and Telecommunications Research Institute) project with the name of *Reliability Improvement of Switching Software*

0-8186-8271-X/97 $10.00 © 1997 IEEE

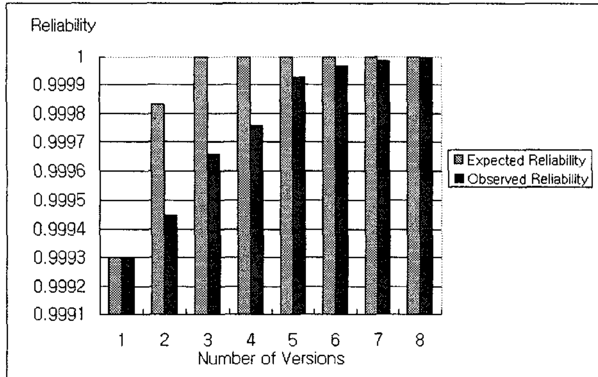

Figure 1. Expected Reliability v.s. Actual Reliability of N-Version Systems

periments, 13 out of 18 fault pairs known to cause CMF in example programs were detected.

In Section 2, the effects of CMF for an NVS reliability and existing NVS development methods are examined. In Section 3, we present a detailed description of our proposed method. An experimental application of proposed method to existing programs is described in Section 4. Conclusions are given in Section 5.

2. Research Backgrounds

A general model for the probability P_N that a majority of the components fail in a system of N versions ($N = 1, 3, 5, \ldots$) is

$$P_N = \int_{[0,1]} \sum_{l=m}^{N} \binom{N}{l} [\theta(x)]^l [1-\theta(x)]^{N-l} dQ(x), \quad (1)$$

where $m = (N+1)/2$ and $\theta(x)$ is the probability that a program, randomly chosen out of a population of versions, will fail on a particular input x [7, 8]. If the versions were to fail independently, that is, $\theta(x)$ does not vary with different inputs, Eq. (1) yields an estimator of P_N given by

$$\overline{P}_N = \sum_{l=m}^{N} \binom{N}{l} \hat{p}^l (1-\hat{p})^{N-l} \quad (2)$$

where $\hat{p}$ is an estimate of the failure probability of a single version. Difference between Eq. (1) and Eq. (2), reliability degradation due to CMF, can be seen from the experimental data of [12]. The average failure rate of 27 versions used for the experiment was 0.0007 [12]. Substituting $\hat{p}$ with 0.0007 and l with a number, the probability that l versions fail simultaneously when the independent failure assumption holds(F_{Expected}) can be derived. Fig. 1 shows the expected reliability(1-F_{Expected}) and the observed reliability(1-F_{Observed}) of N-version systems. It is clear from Fig. 1 that the actual N-version systems have higher failure rates than when the independent failure assumption holds. Furthermore, the effects of CMF are greater when the NVS is composed of relatively small number of versions(Fig. 1). Considering that it is extremely rare in an industrial application of NVP to develop more than 3 versions due to high development cost, there must be some means to reduce the effects of CMF to further improve the reliability of N-version systems.

Several approaches to the development of NVS have been proposed to maximize the diversity among versions to reduce the possibility of CMF. The diversity encompasses different algorithms, programming languages, environments, implementation techniques and tools [2]. Kelly proposed to use more than one specification derived from the same set of user requirements [10], and Avizienis proposed to use different programming languages for each version [2]. Lyu proposed an NVP development method called "N-version Software Design Paradigm" which defines strict rules for independent development of component versions [18].

However, whether proposed NVP development methods were successful in effectively avoiding CMF is in question; independently performed experiments showed that CMF do occur despite all these efforts [12, 8, 13].

3. Detecting Common Mode Failures

A version of an NVS can fail at the same time as other versions in two different ways. It can fail with the same outputs as other versions or with different outputs. The versions failing with different outputs are less harmful than versions failing with the same outputs because the different results produced by failing versions can be detected and some actions can be taken to decrease the effect of incorrect outputs. In this paper, we concentrate on CMF caused by versions failing with the same incorrect output.

Testing has been the primary mean to detect CMF in previous studies related to NVP. For example, a *gold program*, which had been subject to extensive analysis, was used as an oracle to test CMF in the experiment of [12]. The output of each version was compared to that of the gold program and the mismatch was judged to be a failure of a tested version. But in later studies using the same program as [12], additional failures were identified [15]. This incompleteness is inherent in testing because it is impractical to test a program for all possible inputs. Testing is one of forward approaches in which subsets of inputs are chosen and then output spaces that can be generated by these inputs are verified against some kind of oracles.

Forward approaches like testing may not be effective for the CMF detection. Fig. 2 represents a general model

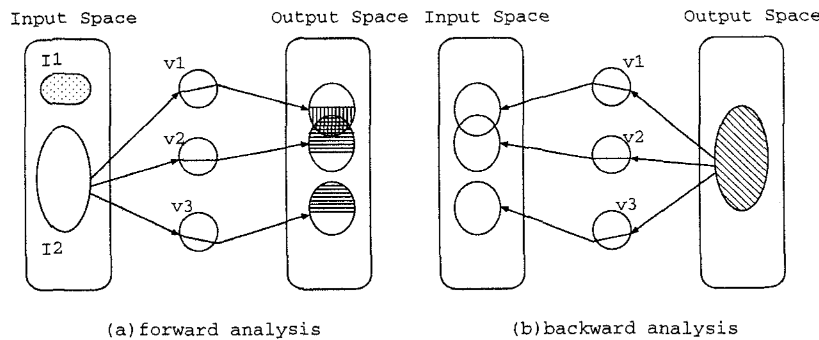

Figure 2. Detection of Common Mode Failures

of the forward analysis to detect CMF. In Fig. 2 (a), three versions($v1, v2, v3$) are analyzed for a possibility of CMF. If each version is analyzed for a subset of input space $I2$, and the shaded areas are the cases where incorrect output values occur, then $v1$ and $v2$ have CMF. On the other hand, $v1$ and $v3$ do not have CMF as far as $I2$ is concerned. In the forward analysis, determining which subsets of input spaces to analyze is the most important and difficult step. Subsets of the input space to be examined are determined mainly by the analyst's experience and domain knowledge. Operational profiles from a real environment in which the system will operate can be used also. Inappropriate choices of the input subsets may result in incomplete analysis. Suppose that in Fig. 2 (a), input space subsets $I1$ and $I2$ may cause CMF but $I1$ is not selected for analysis. Then analysis results may be incomplete, that is, $v2$ and $v3$ may have CMF for input space subset $I1$ but only CMF between $v1$ and $v2$ are detected.

Detection of CMF can be done backward, where failure conditions are determined first and analyzed backward to see if there exist some input space subsets that result in program states satisfying failure conditions. In Fig. 2 (b), as the result of backward analysis for the given failure condition, $v1$ and $v2$ have common failure modes, whereas $v3$ has not with other versions for the given failure condition.

As in the case of forward analysis, determining which subsets of output space to analyze is a crucial step for an effective analysis, and it is not possible to analyze the given programs against every possible failure. However for safety-critical systems, where NVS is frequently employed, results of hazard analysis can be used as the primary target of backward analysis. Backward analysis can increase confidence in the system safety by showing that there is no CMF leading to a mishap [17].

Software fault tree analysis is a well known backward analysis method [14]. In software fault tree analysis, a program state is analyzed using Boolean logic. The root of a fault tree is the event to be analyzed. Children of the root are the causes of the root event. Two events are conjoined(Boolean AND) if both are required to cause the parent event. They are disjoined(Boolean OR) if any one of the event can cause their parent event. Those events perceived to be capable of causing the top-most fault are themselves broken down into their component causes, and this process is repeated until the basic events are reached or it is decided that the events are unanalyzable [14].

The weakest precondition concepts are similar to the software fault tree analysis in that from output spaces, input spaces that can result in given output spaces are analyzed. In fact, software fault tree analysis is a graphical representation of the weakest precondition concepts [16] and the weakest precondition analysis is more formal than the software fault tree analysis. In software fault tree analysis, the analyst can arbitrarily declare an event as infeasible and stop further analysis.

The weakest precondition of a program for a given predicate(*postcondition*) is a predicate representing program states which guarantees the program termination and will result in program states that satisfy the postcondition after the program terminates [6]. For example, if the statement is "*if* $x \geq y$ *then* $z{:}=x$ *else* $z{:}=y$", and the postcondition is "$z{=}y$" then $wp(S, R) = y \geq x$. With $wp(S, R) = y \geq x$ we can be sure that the execution of the *if* statement will terminate resulting in a program state satisfying "$z{=}y$" whenever the execution begins from a program state satisfying $y \geq x$. In this paper, we will use the weakest precondition analysis to detect CMF.

A version fails when an input results in a specific output that does not comply to the specification. Let P_I be a predicate representing the input space, P_O a predicate representing the output space, P_{EI} and P_{EO} predicates representing subsets of input and output spaces related to the failure, respectively. Then failures can be modeled as $P_{EI} \wedge P_{EO}$. However a predicate of the form $P_{EI} \wedge P_{EO}$ cannot be used for analysis of programs in general; it requires the complete information about subsets of input space that can result in failures. It is not possible to decide P_{EI} completely in general. In this paper, we will specify a postcondition in the

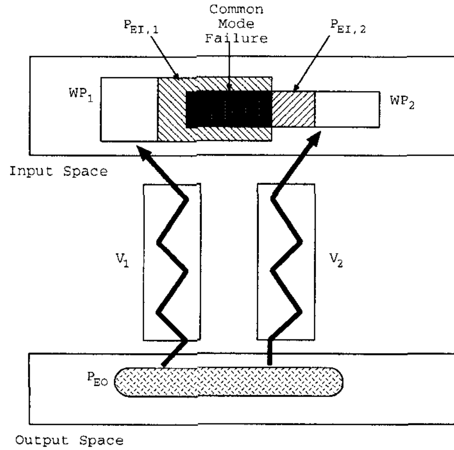

Figure 3. Postcondition, Preconditions and Common Mode Failure

form of P_{EO}.

step 1. Compose P_{EO} from system hazard analysis information

step 2. For each version V_i do
step 2.1 Compute the weakest precondition WP_i
step 2.2 Compare WP_i to the specification and identify $P_{EI,i}$

step 3. Check if $\bigwedge_{\forall i} P_{EI,i}$ is not null

step 4. If $\bigwedge_{\forall i} P_{EI,i}$ is not null
give this subset of input space the highest priority in testing and debugging

Figure 4. Steps of Applying the Weakest Precondition Analysis

Fig. 4 shows the steps of the proposed method. In step 1, a postcondition P_{EO} is specified. This step can be done according to several guidelines. The postcondition can be specified based on the analyst's insights or experiences. It can be specified utilizing the results from system hazard analysis also. As previously mentioned, the postcondition consists of conditions about output variables. In step 2, each version is analyzed for CMF with the postcondition specified in step 1. Step 2 can be divided into two minor steps. First, the version under consideration(V_i) is deployed for the weakest precondition analysis with the given postcondition. As we compose the postcondition out of output variables, each derived weakest precondition(WP_i) should be compared to the specification to see if there exist some parts of the weakest precondition that do not comply to the specification. The identified mismatched parts($P_{EI,i}$) are then compared to each other in step 3. The conjunction of all $P_{EI,i}$ or some of them represents input conditions that result in CMF of conjoined versions. In step 4, the identified CMF conditions, $\bigwedge_{\forall i} P_{EI,i}$, are utilized for further testing and debugging.

Fig. 3 graphically shows these 5 steps for version $1(V_1)$, version $2(V_2)$, and the postcondition P_{EO}. As step 2 is completed, the weakest preconditions WP_1 and WP_2 are derived. Then $P_{EI,1}$ and $P_{EI,2}$ are identified not to comply to the specification. $P_{EI,1} \wedge P_{EI,2}$ is the input space that incorrectly results in P_{EO} simultaneously.

4. A Case Study:Launch Interceptor Program

The Launch Interceptor Program(LIP) is a simple but realistic anti-missile software that was used in several experiments related to NVP technique [12, 15]. LIP is required to check launch interceptor conditions(LIC) for given data representing radar reflections. Twenty seven versions were developed by independent programmers. The existence of CMF in 27 versions was verified in [12] and 45 faults were identified in [5]. Most of the failures were caused by a single fault, but some of them by more than 2 faults. Then, each fault pair was statistically tested if they cause CMF. Ninety three fault pairs out of 945 fault pairs were identified to cause CMF and 67 fault pairs were suspected to cause CMF but the number of test samples was not large enough to be statistically valid. Identified faults could be classified as logical faults and faults related to the precision of arithmetic operations [5]. The arithmetic operations performed in a finite precision give rise to consistent comparison problems [4]. The weakest precondition analysis, a logical analysis method, is inadequate to analyze these finite precision-related faults. We excluded these faults in this analysis by assuming that all numerical operations are carried out with a mathematical precision.

Among 27 versions and 15 LIC's, LIC 3 of versions 3, 8, 20, 25 were analyzed in this paper. These versions were chosen because they have a relatively small number of faults related to the finiteness of numerical operations. LIC 3 reads as follows [12](Fig. 5).

> "3) There exists at least one set of three consecutive data points which form an angle such that: $angle < (\pi - \epsilon)$ or $angle > (\pi + \epsilon)$. The second of the three consecutive points is always the vertex of the angle. If either the first point or the last point(or both) coincides with the vertex, the angle is undefined and the LIC is not satisfied by those three points. $(0 \leq \epsilon < \pi)$"

Faults of analyzed versions identified in [5] are listed in table 1.

FAULT	LIC#	INPUT CONDITION	FAULT DESCRIPTION
3.1	LIC 3	Three collinear points (subtended angle zero)	Path that handles collinear points always gives π as subtended angle - misses case in which angle is zero.
3.4	LIC 3,10	Three almost collinear points	Inaccurate algorithm to determine collinearity; points treated as collinear when just nearly collinear.
8.1	LIC 3,10	Three collinear points (subtended angle zero)	Similar to fault 3.1.
8.2	LIC 3,10	Three collinear points (subtended angle zero)	Similar to fault 8.1 but on special path to handle horizontal and vertical lines
20.1	LIC 3,10	Three collinear points (subtended angle zero)	Similar to fault 3.1.
20.2	LIC 3,10	Three collinear or almost collinear points (subtended angle zero)	In applying formula tan = sqrt(1-sqr(cos))/cos, roundoff error causes negative argument to sqrt
25.1	LIC 3,10	Three collinear points (subtended angle zero)	Missing case in computing angle formed by three points when point 1 lies between points 2 and 3 and is closer to point 2 than to point3.
25.2	LIC 3,10	Three collinear points (subtended angle zero)	Similar to fault 25.1 different path

Table 1. Faults of Analyzed Versions

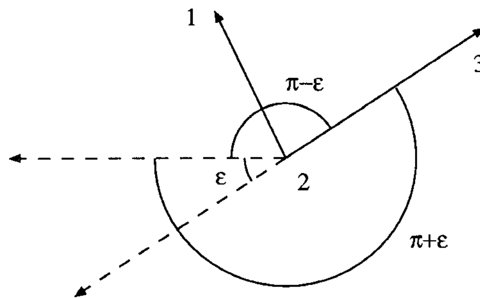

Figure 5. Requirements of LIC3

To apply the weakest precondition analysis to LIP versions, we modified original programs in two ways. Original programs use `realcompare(x,y)` to allow tolerance when comparing two real numbers with finite precision. Since we assumed a mathematical precision for numerical operations, comparison operations allowing tolerance is not needed. For example,

```
...
lic3cond :=
   (realcompare(alpha,pi-repsilon)=lt)
   or
   (realcompare(alpha,pi+repsilon)=gt);
...
```

was changed into

```
...
lic3cond := (alpha<pi-repsilon) or
            (alpha>pi+repsilon)
...
```

Second, we eliminated loops over input arrays. LIP versions are required to check each LIC over all possible input subsequences. LIP versions do this by checking each LIC for an input set and repeat the checking process after changing the input set by modifying indices to input array. For example,

```
repeat
  j := i + 1;
  k := j + 1;
  ...
    if equalpoints (i, k)
       then ... ;
    ...
  i := i + 1;
until (i >= maxloop) or lic3cond ;
```

checks whether $inputarray[i]$ is the same point as $inputarray[k]$ varying indices of $inputarray$ to $maxloop$. We changed this code into

```
...
if equalpoints (x1,y1,x3,y3)
     then ... ;
...
```

eliminating the loop, and modifying variables appropriately. The listings of the modified program against which we applied our method can be found in the Appendix.

In this paper, we used *LIC3=False* as the postcondition. As it is not a complete specification of failures but a condition about the output variables, the weakest preconditions attained through the analysis should be examined against the LIC3 specification.

Let (xi, yi) denotes coordinates of ith point and A, B and C denote the distances between two points as shown in Fig. 6. Then, version 3 fails with output *False* when $(x1 \neq x3 \vee y1 \neq y3) \wedge (A + C = B \vee B + C = A)$. We applied the weakest precondition analysis to version 8

also and identified subsets of the input space that incorrectly result in $LIC3 = False$ for version 8.

- When $(x1 \neq x3 \wedge y1 = y2 \wedge y2 = y3 \wedge (\neg(x1 < x2 < x3) \vee \neg(x3 < x2 < x1))) \vee (y1 \neq y3 \wedge x1 = x2 \wedge x2 = x3 \wedge (\neg(y1 < y2 < y3) \vee \neg(y3 < y2 < y1)))$. When three points form a horizontal or vertical line and the second point does not lie in the middle of other points, the angle formed by three point is 0. The output should be *True*.
- When $(x1 \neq x3 \vee y1 \neq y3) \wedge (x1 \neq x2 \wedge x2 \neq x3) \wedge \frac{y1-y2}{x1-x2} = \frac{y3-y2}{x3-x2}$. Two lines have the same slopes resulting in the angle of size 0, thus the output should be *True*. It is an incorrect output.

From these results it can be found that version 3 incorrectly sets its output as *False* when three points are on a line and the distance between $(x1, y1)$ and $(x3, y3)$ is not the longest of all distances and version 8 incorrectly sets its output as *False* when three points are on a line and $(x2, y2)$ is not in the middle of three points. These cases represent the same condition and versions 3 and 8 have CMF.

We analyzed other two versions also and identified conditions that result in CMF for each version pair. Version 20 had the same CMF as versions 3 and 8. It produced the incorrect *False* output when the angle formed by three points was zero. Version 25 produced the incorrect *False* output when the angle formed by three points were zero and A is equal to or longer than B. This condition is a subset of CMF condition between other three versions.

We located the faults causing specified failures using identified incorrect input conditions as explained in the step 4 in Fig. 4. We performed the program reading with an input satisfying identified incorrect input conditions. Other testing and debugging methods could be also used, but for this example, program reading was sufficient. Fault 3.1 is related to lines 20 and 21 of version 3. The condition part of the *if* statement should be refined for this version to be correct. Fault 8.1 is related to line 38 of version 8. The angle defined should be zero rather than π. Fault 8.2 is similar to fault 8.1 but related to line 24. Line 24 handles special cases of three points forming a horizontal or vertical line. These

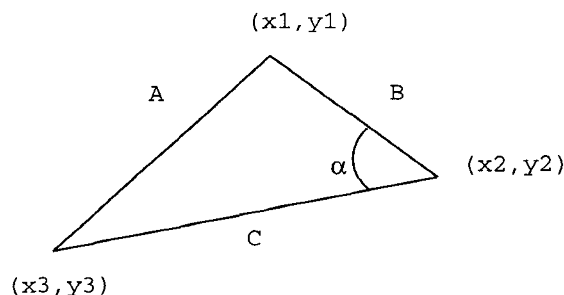

Figure 6. Angle formed by three points

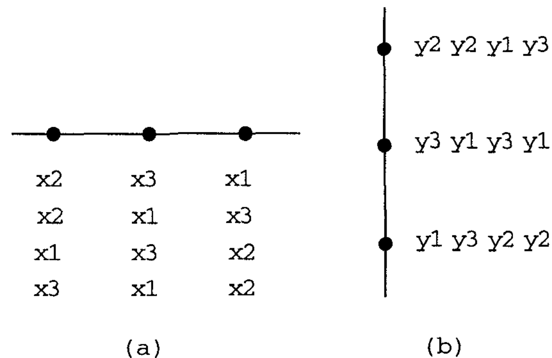

Figure 7. Three points form a horizontal or a vertical line

results are summarized in table 1. We identified faults 3.1, 8.1, 8.2, 20.1, 25.1 and 25.2 of table 1.

Table 2 compares our analysis results with the previously identified fault pairs causing CMF. There are 24 fault pairs that are capable of causing CMF in 4 versions. Among them, 6 fault pairs were identified not to cause CMF in the previous analysis and also in our analysis. We identified 13 of the rest of the fault pairs to cause CMF, and 5 fault pairs not found by our analysis contains faults related to finite precision numerical operations.

5. Conclusions

Several approaches for NVS development have been proposed to prevent CMF and thus enhancing the overall reliability, but they failed to prevent CMF completely. After all, it is unlikely that a single method can prevent CMF completely.

In this paper, we proposed to detect CMFs related to the more important properties than to try to remove all possible CMFs in the system using the weakest precondition analysis. Beginning with a predicate described in terms of output

	3.1	3.4	8.1	8.2	20.1	20.2	25.1	25.2
3.1	-							
3.4	-	-						
8.1	v	.	-					
8.2	v	.	-	-				
20.1	v	.	v	v	-			
20.2	m	.	m	m	-	-		
25.1	v	.	v	v	v	m	-	
25.2	v	.	v	v	v	m	-	-

(v: causing CMF and found by analysis, m:causing CMF but not found, -: not applicable, .: not causing CMF and related to the precision of numerical operations

Table 2. Summary of Analysis Results

variables, we identified input regions that could result in outputs satisfying the given predicate through the weakest precondition analysis. Each identified input region was then compared to the specification and subsets of input spaces that cause CMF were identified. Our experimental application of the proposed method to a existing realistic example program, the Launch Interceptor Program, showed its effectiveness in detecting CMFs related to logical flaws in the program.

As the development cost is a prohibiting factor for a wider application of NVP, the cost of applying the proposed method might be problematic; the cost of applying our method is inevitable to some extent. The diversity required for each version of the NVP implies that the actual processes of the proposed analysis method do not have much in common, resulting in a high cost. However, this cost can be reduced. The proposed method is a backward approach, and it can be applied according to the importance of the failure modes. When the cost is important, it can be applied to a limited number of properties of the system. For example, hazard states of safety-critical systems are the primary target of CMF analysis. The labor-intensive part of the overall process can be minimized if an interactive tool is utilized for some simple analysis rules.

A proper specification of a postcondition is another obstacle for a effective application of the proposed method. The postcondition is a condition about outputs and an improper postcondition might lead to an ineffective analysis results. For safety critical systems, the results of system the hazard analysis might be used as a basis for specifying the postcondition and formalization of transforming results of system hazard analysis to postconditions might be helpful in applying the proposed method.

References

[1] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Execution," *Proc. IEEE COMPSAC '77,* pp. 149-155, Nov. 1977.

[2] A. Avizienis, M. R. Lyu, and W. Schutz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," 18th International Symposium on Fault Tolerant Computing, 1988.

[3] P. Bishop, "software Fault Tolerance by Design Diversity," In *Software Fault Tolerance,* Edited by M. R. Lyu, pp. 211-229, John Wiley & Sons Ltd, 1995.

[4] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "The Consistent Comparison Problem in N-version Software," *IEEE Trans. Software Eng.*, vol. 15, pp. 1481-1485, Nov. 1989.

[5] S. S. Brilliant, J. C. Knight and N. G. Leveson, "Analysis of Faults in an N-Version Software Experiment," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 238-247, Feb. 1990.

[6] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *CACM*, vol. 18, no. 8, pp. 453-457, Aug. 1975.

[7] D. E. Eckhardt and L. D. Lee, "A Theoretical Basis for the Analysis of Redundant Software subject to Coincident Errors," NASA Langley Research Center, Hampton, VA, Rep. NASA TM 86369, Jan. 1985

[8] D. E. Eckhardt , A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk and J. P.J. Kelly, "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," *IEEE Trans. Software Eng.*, vol. 17, pp. 692-702, July 1991.

[9] D. Gries, "The Science of Programming," Springer-Verlag New York Inc., 1981

[10] J. P. J. Kelly and A. Avizienis, "A Specification Oriented Multi-Version Software Development," In *Digest of 13th FTCS,* pp. 120-126, Jun. 1983.

[11] J. P. J. Kelly and S. C. Murphy, "Achieving Dependability Throughout the Development Process: A Distributed Software Experiment," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 153-165, 1990.

[12] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Software Eng.*, vol. 12, No. 1, pp. 96-109, Jan. 1986

[13] J. C. Knight and N. G. Leveson, "A Reply to the Criticisms of the Knight & Leveson Experiment," *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 1, pp. 24-35, Jan. 1990.

[14] N. G. Leveson, P. R. Harvey, "Software Fault Tree Analysis," *Journal of Systems and Software*, Vol. 3, No. 2, pp. 173-181, 1983.

[15] N. G. Leveson, S. S. Cha, J. C. Knight and T. J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study," *IEEE Trans. Software Eng.*, vol. 16, no. 4, pp. 432-443, Apr. 1990.

[16] N. G. Leveson, S. D. Cha and T. J. Shimeall "Safety Verification of Ada Programs using Software Fault Trees," *IEEE Software*, Vol. 8, No. 4, July 1991.

[17] N. G. Leveson, *Safeware : System Safety and Computers*, Addison-Wesley Pub. Co., 1995.

[18] M. R. Lyu and A. Avizienis, "Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming," pp. 197-218. In J.F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2,* Springer-Verlag, 1992.

A. Modified Program Listings

The modified program of version 3 follows.

```
 1     function lic3: boolean;
       ....
 4     begin
 5        lic3 := false;
          ....
11        if ( (x1=x2 and x1=y2) or (x2=x3 and y2=y3) )
12        then lic3 := false
13        else begin
14      if ( x1=x3 and y1=y3 )
15      then alpha := 0
16      else begin
17          a := pointdistance (x1,y1, x3,y3);
18          b := pointdistance (x1,y1, x2,y2);
19          c := pointdistance (x2,y2, x3,y3);
20          if ( a+b=c or a+c=b or b+c=a )
21          then alpha := pi {* FAULT 3.1 *}
22          else begin
23        s := 0.5 * (a + b + c);
24        r := sqrt (((s-a)*(s-b)*(s-c))/s);
25        alpha := 2 * arctan (r/(s-a));
26          end; {else}
27      end; {else}
28      if (alpha<pi-epsilon) or (alpha>pi+epsilon) then
29          lic3 := true;
30        end;
31   end;

function pointdistance (x1,y1,x2,y2: real): real;
var xd, yd: real;
begin
    xd := x2 - x1;
    yd := y2 - y1;
    pointdistance := sqrt ((xd*xd) + (yd*yd));
end;
```

The modified program of version 8 looks as follows.

```
 1  function lic310():boolean;
    ...
13  begin
14      found:=false;
        ...
19      if not (((x1=x2) and (y1=y2)) or ((x2=x3) and (y2=y3)))
20      then begin
21          if ((x1=x3) and (y1=y3))
22          then testangle:=0.0
23          else if (((x1=x2) and (x2=x3)) or ((y1=y2) and (y2=y3)))
24          then testangle:= pi       {* FAULT 8.2 *}
25          else
26          begin
```

```
27        if not ((x1=x2) or (x2=x3))
28        then begin
29            slope1:=(y1-y2)/(x1-x2);
30            slope2:=(y3-y2)/(x3-x2);
31        end
32        else begin
33            slope1:=111;
34            slope2:=999
35        end;
36
37        if slope1=slope2
38        then testangle:=pi    {* FAULT 8.1 *}
39        else
40        begin
41            distances(x1,x2,x3,y1,y2,y3,
42                  a,b,c);
43            asqr:=sqr(a);
44            bsqrcsqr:=sqr(b) + sqr(c);
              ...
46            if asqr=bsqrcsqr
47            then testangle:=pi/2.0
48            else
49            begin
50         findangles(x1,x2,x3,y1,y2,y3,
51                     a,b,c,semi,sina,cosa,rsina,rcosa);
52         if asqr<bsqrcsqr
53         then testangle:=arctan(rsina/rcosa)
54         else testangle:=pi - arctan(rsina/rcosa)
55            end
56        end
57           end;
58           if (testangle<(pi-epsilon)) or
59        (testangle>(pi+epsilon))
60           then found:=true
61       end
         ...
64   end;

procedure findangles(x1,x2,x3,y1,y2,y3,a,b,c,semi,sina,cosa:real;
              var rsina,rcosa:real);
begin
  semi:=(a+b+c)/2.0;
  sina:=2.0/(b*c) * sqrt(semi*(semi-a)*(semi-b)*(semi-c));
  cosa:=sqrt(1.0-sqr(sina));
  rsina:=sina* pi/180.0;
  rcosa:=cosa* pi/180.0
end;
```