N-VERSION PROGRAMMING : A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION

Liming CHEN
Xerox Corporation
El Segundo, CA 90245 USA

Algirdas AVIZIENIS
Computer Science Department
University of California
Los Angeles, CA 90024 USA

Abstract

N-version programming is defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. A methodology of N-version programming has been devised and three types of special mechanisms have been identified that are needed to coordinate the execution of an N-version software unit and to compare the correspondent results generated by each version. Two experiments have been conducted to test the feasibility of N-version programming. The results of these experiments are discussed. In addition, constraints are identified that must be met for effective application of N-version programming.

1. Approaches to Software Fault-Tolerance

The usual method to attain reliability of software operation is fault-avoidance (or intolerance) [1]. All software defects are eliminated prior to operation. If some defects remain, the operation is reliable only as long as the defects are not involved in program execution. In most large and complex software systems these fault-avoidance conditions have not been successfully attained, regardless of a very large investment of effort and resources, and software crashes have occurred during operation. This observation leads to the conjecture that for reliable software operation, redundant software in some form is required to detect, to isolate, or to recover from effects of the thus far uneliminated software defects.

Achievement of high reliability of operation through the use of redundant system elements is a fundamental principle in fault-tolerance of hardware (physical) faults [4]. The use of redundant software to recover from software malfunction, however, requires special caution due to the idiosyncratic characteristics of software. In contrast with hardware, in which physical faults predominate, software defects are time-invariant defects. Errors are produced by using the same inputs which trigger the same deficient elements of a program. Therefore, executing duplicate copies of a program does not improve the reliability of operation with respect to software defects. Furthermore, while the main cause of hardware unreliability is a random failure,that of software is its complexity. The complexity of software leads to several difficulties. First, it is difficult to construct error-free software. Second, software is unlikely to perform complete self-checking on its own outputs. Third, it is difficult to perform run-time diagnosis of software in order to locate the source of a software error. These observations lead to the conclusion that if redundant software is used in an attempt to achieve software fault-tolerance, then it should try to meet the following constraints: (1) It does not require complete self-checking; (2) it does not rely upon run-time software diagnosis; and (3) it must contain independently developed alternative routines for the same functions.

Experience with fault-tolerance of hardware (physical) faults suggests that functionally equivalent alternative routines may be employed to improve reliability of software operation. Recently, two distinct approaches have been investigated which employ alternate software routines as a means to achieve software fault-tolerance. In the approach of recovery blocks [2, 3] these routines are organized in a manner similar to the dynamic redundancy (standby sparing) technique in hardware [4]. The prime objective is to perform run-time software error detection and to implement error recovery by taking an alternate path of operation.

A potential alternative to recovery blocks is to use software redundancy analogously to the static (replication and voting) redundancy approach in hardware [4]. The prime objective here is to mask the effects of software defects at the boundaries of designated program modules. The first technical discussion of this approach in which one of the authors took part occurred in February 1966 at the IEEE Workshop on the Organization of Reliable Automata in Pacific Palisades, Ca. Several suggestions that this approach might be a viable method of software fault-tolerance were published a few years later [1, 5, 6, 7, 8]. In 1975, an experimental research project entitled, "N-Version programming" was initiated at UCLA to systematically investigate the feasibility of this approach [1, 9, 10].

2. Concepts of N-Version Programming

N-version programming is defined as the independent generation of $N \geq 2$ functionally equivalent programs, called "versions", from the same initial specification [9]. (This term is preferred to "distinct software," [8], since it bears no implication about the "distinctness," which is vague and difficult to quantify or even qualify, among the N versions of a program.) "Independent generation of programs" here means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages or translators are used in each effort.

The initial specification is a formal specification in a specification language. The goal of the initial specification is to state the functional requirements completely and unambiguously, while leaving the widest

Reprinted from *FTCS-8* 1978, pp. 3-9, © IEEE
0-8186-7150-5/96 $5.00 © 1996 IEEE
Proceedings of FTCS-25, Volume III

possible choice of implementations to the N programming efforts. It also states all the special features that are needed in order to execute the set of N version in a fault-tolerant manner. An initial specification should define: (1) the function to be implemented by an N-version software unit; (2) data formats for the special mechanisms: comparison vectors ("c-vectors"), comparison status indicators ("cs-indicators"), and synchronization mechanisms; (3) the cross-check points ("cc-points") for c-vector generation; (4) the comparison (matching or voting) algorithm; and (5) the response to the possible outcomes of matching or voting. We note that "comparison" is used as a general term, while "matching" refers to the N = 2 case, and "voting" to a majority decision with N > 2. The comparison algorithm explicitly states the allowable range of discrepancy in numerical results, if such a range exists.

N versions of the program are independently generated with respect to the initial specification. Though different in their implementation, the N versions are assumed to be functionally equivalent. Together, these N versions are said to form an N-version software unit. During the development of an N-version program, the performance of each version must satisfy some acceptance criteria of its own before it can be integrated into the N-version software unit. An acceptance program can be used to drive a single version for its acceptance testing. To drive an N-version software unit a supervisory program called a driver, is neeeded. It is a modified acceptance program with additional capabilities to coordinate the execution of N versions and to vote or to match their correspondent results. The integrated set of an N-version software unit and its driver is said to be an N-version program.

Three types of special mechanisms are needed to execute an N-version software unit and to match or vote the correspopndent results generated by each version. These special mechanisms are: (1) comparison (c-) vectors, (2) comparison status (cs-) indicators, and (3) synchronization mechanisms. The points at which c-vectors are generated and employed for matching or voting are called cross-check (cc-) points.

C-vectors are data structures representing a subset of a version's local program state which is interpretable by the driver. Meaningful interpretation of a c-vector, however, can only be achieved when a cc-point condition has been satisfied. A c-vector generated by a version at a cc-point contains two types of information. The comparison variables (c-variables) of a c-vector point to values of variables which are to be matched with their counterparts from other versions. The status flags of a c-vector indicate whether or not some significant events have taken place during the generation of these c-variables. Examples of such events are: end of file, exception conditions detected by the system, or conditions defined in the initial specification. When majority of versions produces the results that agree (i.e, fall within the allowable range of discrepancy), these results are treated as acceptable results from this N-version software unit. Any version which generates results that differ from the acceptable results is designated as a disagreeing version.

Cs-indicators are used to indicate actions to be taken after matching or voting of correspondent c-vectors when a cc-point condition is satisfied. The actions to be taken at the cc-points after the exchange of c-vectors depend on: (1) whether all versions deliver the c-vectors within specified time, and (2) whether the c-vectors agree or disagree. Possible outcomes are: (1) continuation, (2) termination of one or more versions, and (3) continuation after changes in the c-vectors of one or more versions on the basis of a majority decision.

Synchronization mechanisms are used to synchronize the execution steps of an N-version software unit. Each version uses these mechanisms to signal to the driver that a c-vector is ready. The driver uses these mechanisms to control when a version should be activated. They are also used by the driver to prevent voting or matching before all correspondent c-vectors are ready. Originally, a version is in an inactive state. When invoked by the driver, it enters into a waiting state. At this state it waits for a synchronization signal representing a request for service from the driver. When this signal is received, it transfers into a running state. If any terminating condition is signaled by the cs-indicators, then the execution of this version is terminated and it goes back to inactive state. Otherwise, it generates a c-vector upon the satisfaction of a cc-point condition, then it uses a synchronization signal to notify the driver that a c-vector is ready, and then returns to waiting state. The state transitions for a version are illustrated in Figure 1. It should be noted that state transitions due to system resource allocation and deallocation are not of direct concern to N-version programming and are not discussed here.

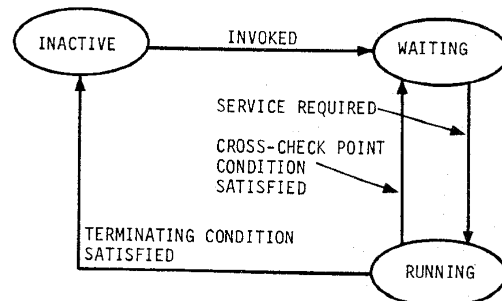

Figure 1 State Transitions of a Version

A limitation of the N-version approach results from the fact that all N versions of the program originate from the same initial specification, which is effectively the "hard core" of this method. Its correctness, completeness, and unambiguity have to be assured prior to the N-version programming effort. It is our conjecture that either formal correctness proofs, or exhaustive validations of initial specifications when they are stated in compact, formal specification languages [11] are much more likely to succeed within acceptable cost bounds than proofs or validations of the detailed implementations that originate from such specifications. Once the specifications have been accepted as correct, the proofs or validations of the programs can be replaced by the run-time software fault-tolerance provisions.

The second major observation concerning N-version programming is that its success as a method for run-time tolerance of software faults depends on whether the residual software faults in each version are distinguishable. We define distinguishable software faults as faults that will cause a disagreement between c-vectors at the specified cc-points during the execution of the N-version program that was generated from the initial specification. Distinguishability is affected by the choice of c-vectors and cc-points, as well as by the nature of the faults themselves. It is a fundamental conjecture of the N-version approach that the independence of programming efforts will greatly reduce the probability of identical software defects

occuring in two or more versions. Together with a reasonable choice of c-vectors and cc-points this is expected to turn N-version programming into an effective method to achieve tolerance of software faults. The effectiveness of the entire approach depends on the validity of this conjecture, therefore it is critically important to keep the initial specification free of any flaws that would bias the independent programmers toward introducing the same software defects.

The research effort at UCLA addresses two thus far unanswered questions: (1) Which constraints (e.g., need for formal specifications, suitable types of problems, nature of algorithms, timing constraints, etc.) have to be satisfied to make N-version programming feasible at all regardless of the cost? (2) How does the cost-effectiveness of the N-version programming approach compare to the two alternatives: non-redundant ("fault-intolerant") programming [1], and the "recovery block" [2, 3] approach? The scarcity of previous results and an absence of formal theories on N-version programming has led us to choose an experimental approach in this investigation. The approach has been to choose some conveniently accessible programming problems, to assess the applicability of N-version programming, and then to proceed to generate a set of programs. Once generated, the programs are executed in a simulated multiple-hardware system, and the resulting observations are applied to refine the methodology and to build up theoretical concepts of N-version programming. A more detailed discussion of the research approach and goals can be found in [9], and a detailed discussion of experimental results in [10].

3. A Comparison of Approaches

In comparison to N-version programming the recovery block approach has one apparent advantage. In some situations, a software system evolves by replacement of some of its modules with newly developed ones. The replaced modules can be used as supplementary alternates to the new modules. The production cost is lower in this case.

However, there are also certain disadvantages associated with the recovery block approach. First, the system state before entry into a recovery block must be saved until some reasonable results are obtained from the block. Considerable storage overhead may then be involved for nested recovery block structures. Second, special precautions are needed to coordinate parallel processes within a nested recovery block structure. Otherwise the interdependencies among these processes may require that a long chain of process effects should be undone after a process has failed [2]. Third, some intermediate output from a recovery block may not be reversible in a real-time environment. Therefore, no recovery action can be performed before the incorrect output causes its damage. Fourth, special system support is necessary to alleviate the above weaknesses. This limits the generality of applications of the recovery block technique.

Finally, we also note that the effectiveness of the acceptance test is often quite difficult to measure. In many cases, the procedure used to verify results from the execution of a program can be as complex as the program itself. For example, it is easy to check the consistency of the number of elements in a set before and after execution of a sorting routine. It is more difficult to verify that all of the data items are indeed sorted as specified. It is even more difficult to verify that the elements of the set before and after the sorting are the same. Therefore, it is obvious that in most cases only "reasonableness" rather than "correctness" may be checked for by acceptance tests. The lack of established procedures to estimate the effectiveness of acceptance tests leaves it hard to determine if it is sufficient to use a given acceptance test for a very critical application.

In view of the above difficulties, the N-version programming approach offers some advantages over the recovery blocks. In this approach, self-checking is not required. Some redundant software can be eliminated; this seems attractive from the coverage point of view [12]. It also offers the possibility of immediately masking some software faults so that there is no delay in operation.

In certain applications, N-version programming also makes better use of existing hardware fault-tolerance resources. For instance, there are recent system designs for aerospace applications that use redundant hardware at the system level to attain fault-tolerance. The SIFT design [13], the Symmetric Multiprocessor [14] and the central computer complex in the Space Shuttle [15] are some examples. In these systems, copies of identical programs are executed in three or more identical processor-memory units, and voting of the results allows detection and masking of hardware faults. Since full system reliability requires the reliable operation of both hardware and software, these designs are vulnerable to software deficiencies. The adoption of N-version programming may allow such systems to tolerate both hardware and software faults without delays caused by the acceptance testing used in the recovery block approach.

4. Implementation of N-Version Programming

For the reason of convenience, the following discussion will assume that N=3. An extension to N > 3 is quite straightfoward.

4.1 Special Mechanisms

Implementation of special mechanisms (c-vectors, cs-indicators, and synchronization mechanisms), in a 3-version program is illustrated by Figures 2, 3, and 4. The schemata shown in these figures have been written with the PL/I compiler in mind. It should be noted that as a result of emphasis on readability, the length of some identifiers or labels may not be allowed in some implementations.

```
VERSIONi: PROCEDURE OPTIONS (TASK);
   DCL 1 C-VECTORi                      EXTERNAL,
         2 { comparison variables }
         : { status      flags    };
   DCL (DISAGREEi, GOODBYE)  BIT(1) EXTERNAL;
   DCL (SERVICEi, COMPLETEi) EVENT  EXTERNAL;
   DCL FINIS        BIT(1)      INIT('0'B);
   other declarations;
   DO WHILE (¬FINIS);
      WAIT (SERVICEi);
      COMPLETION (SERVICEi) = '0'B;
      IF ¬GOODBYE & ¬DISAGREEi
      THEN CALL PRODUCE;
      ELSE FINIS = '1'B;
      COMPLETION (COMPLETEi) = '1'B;
   END;
   PRODUCE: PROCEDURE;
            produce C-VECTORi;
            END PRODUCE;
   END VERSIONi
```

Figure 2 A Schema for the i-th Version of Code

```
ACCEPTANCE: PROCEDURE OPTIONS (MAIN);
   DCL VERSIONi      ENTRY;
   DCL 1 C_VECTORi  EXTERNAL,
         2 { comparison variables }
         : { status      flag     } ;
   DCL DISAGREEi    BIT(1)      EXTERNAL,
       GOODBYE      BIT(1)      EXTERNAL;
   DCL SERVICEi     EVENT       EXTERNAL,
       COMPLETEi    EVENT       EXTERNAL;
   other declarations;
   COMPLETION (SERVICEi) = '1'B;
   COMPLETION (COMPLETEi) = '0'B;
   CALL VERSIONi TASK EVENT (FINISi);
   DO WHILE (need_more_service);
      WAIT (COMPLETEi);
      COMPLETION (COMPLETEi) = '0'B;
      process C_VECTORi;
      IF ¬need_more_service THEN GOODBYE = '1'B;
      COMPLETION (SERVICEi) = '1'B;
   END;
   WAIT (FINISi);
   END ACCEPTANCE;
```

Figure 3 A Schema for an Acceptance Program

```
DRIVER:      PROCEDURE OPTIONS (MAIN);
   DCL (VERSION1, VERSION2, VERSION3) ENTRY;
   declare (C_VECTOR1, C_VECTOR2, C_VECTOR3);
   DCL (DISAGREE1, DISAGREE2, DISAGREE3, GOODBYE)
       BIT(1) EXTERANL;
   DCL (SERVICE1, COMPLETE1,
        SERVICE2, COMPLETE2,
        SERVICE3, COMPLETE3) EVENT EXTERNAL;
   other declarations;
   initialize (SERVICEi, COMPLETEi) as in ACCEPTANCE;
   CALL VERSION1 TASK EVENT (FINIS1);
   CALL VERSION2 TASK EVENT (FINIS2);
   CALL VERSION3 TASK EVENT (FINIS3);
   DO WHILE (need_more_service);
      WAIT (COMPLETE1, COMPLETE2, COMPLETE3);
      process (C_VECTOR1, C_VECTOR2, C_VECTOR3);
      IF ¬DISAGREE1 THEN COMPLETION(COMPLETE1)='0'B;
      IF ¬DISAGREE2 THEN COMPLETION(COMPLETE2)='0'B;
      IF ¬DISAGREE3 THEN COMPLETION(COMPLETE3)='0'B;
      IF ¬need_more_service THEN GOODBYE='1'B;
      COMPLETION(SERVICE1)='1'B;
      COMPLETION(SERVICE2)='1'B;
      COMPLETION(SERVICE3)='1'B;
   END;
   WAIT (FINIS1, FINIS2, FINIS3);
   END DRIVER;
```

Figure 4 A Schema for a Driver

When Figures 2, 3, and 4 are appliea "i" would be replaced by 1, 2, or 3. VERSIONi is the i-th version of a 3-version software unit, ACCEPTANCE is the acceptance program with respect to VERSIONi and DRIVER represents a driver which exercises a 3-version software unit. C-VECTORi represents a c-vector to be produced by the i-th version. The cs-indicator DISAGREEi shows whether or not C-VECTORi agrees with the correspondent acceptable results. Another cs-indicator, GOODBYE, represents whether or not a normal terminating condition is satisfied. The synchronization primitive, SERVICEi, is used to signal a request from the driver for the service of the i-th version. Another synchronization primitive, COMPLETEi, is used by the i-th version to signal the driver that C-VECTORi is ready.

From Figures 2, 3, and 4, it is evident that the implementation of special mechanisms for N-version programming is relatively simple. This is illustrated by the example in Appendix 1.

4.2 Inexact Voting

For numerical computations, two types of deviations may appear in the results. The first type is an "expected" deviation due to the inexact hardware representation or the data sensitivity of a particular algorithm. The second type is an "unexpected" deviation due to either inadequate design or implementation of an algorithm, or a malfunction of hardware. Either type of deviation may cause results obtained from different numerical algorithms to disagree with each other. The standard voting process which requires that the majority of correspondent results should have exactly the same values to determine an acceptable result is not applicable here. Different voting processes need to be devised to handle voting with non-identical results. These voting processes will be called "inexact voting".

In general, adaptive and non-adaptive voting are two alternatives which may be applied to perform inexact voting. Assume that R1, R2, and R3 are correspondent results used to determine the voted result, R. Then in the approach of adaptive voting, as suggested in [16],

$$R = W1*R1 + W2*R2 + W3*R3,$$

where (1) W1, W2, W3, are weights of R1, R2, R3 respectively; (2) W1, W2, W3 are positive values; and (3) W1+W2+W3=1. These weights may be dynamically calculated based on the values of R1, R2, and R3. The major intent is to favor acceptable results and to minimize the effect of a disagreeing result. In other words, R is constructed to be a continuous function of R1, R2, and R3, that will smooth out the effect of a disagreeing result. To compute the weights of correspondent results, several schemes are available. The performance of a scheme is influenced by its "tolerance" parameter, which is a measure of the allowable noise level and could be optimally determined from the magnitudes of expected results and noisy results.

The adaptive voting approach may suffer from the following disadvantages: (1) The optimal tolerance parameter is difficult to determine unless the characteristics of the expected values and noisy values are known well in advance. (2) The remaining effect of noise may not be acceptable in some cases. (3) If the voted result will be used as input for next cycle of computation then the accumulation of residual effects of noise may cause a serious problem. (4) If implemented in software, the adaptive voter may be quite slow.

As a contrast, the non-adaptive voting approach uses an allowable discrepancy range and differences of pairs of correspondent results in determining R. Assume that δ is the allowable discrepancy range, and Dij is the absolute value of the difference between Ri and Rj. Then an acceptable R may be reached by adopting one of the following two strategies: (1) if maximum (D12, D23, D31) $\leq \delta$ or (2) if minimimum (D12, D23, D31) $\leq \delta$. The first strategy requires D12, D23, D31 be known before R can be determined. In the second strategy, however, if D12 $\leq \delta$ R can be determined without knowing D23 and D31.

The non-adaptive voting approach is not without flaws. First, the value of δ is very difficult to determine dynamically for each instance of voting. Second, the strategy which uses the principle of maximum (D12, D23, D32) $\leq \delta$ is too rigid since an erroneous result may easily cause a Dij which is larger than δ. Acceptable results may not be reached even when two of the three correspondent results are reasonably close. Third, the strategy which uses the principle of minimum (D12, D23, D31) $\leq \delta$ may encounter

situations where each version may have different effects on the outcome of voting. These situations are illustrated better with the following examples. Assume that the allowable discrepancy range is 0.9 and (i) expected R1 = 117.0, (ii) expected R2 = 116.5, and (iii) expected R3 = 115.8. In this case, (i) if only R1 or R3 is erroneous, an acceptable R can still be generated, (ii) but if R2 is erroneous then no acceptable R can be generated.

Therefore, there is no inexact voting approach which can be applied satisfactorily to all cases. The success of an approach usually depends on the designer's knowledge about (1) the data sensitivity of each algorithm, (2) the limitations of hardware representations, and (3) the allowable ranges of discrepancies for each instance of voting.

5. Feasibility Studies of N-version Programming

At an early stage of the investigation of N-version programming, it was decided to conduct a few experiments to gain some insight into the feasibility of this technique. Three objectives were set for the experiments. They were: (1) to study the generality and the ease of the implementation of N-version programming; (2) to gain qualitative and quantitative data on effectiveness of 3-version programming; and (3) to observe and identify problems or difficulties in using 3-version programming.

Three criteria were used to select target problems for feasibility studies. First, a target problem needs to be relatively complex so that there is reasonably good possibility that residual software defects will occur in its implementing programs. Second, the program implementing a target problem should be of manageable size to facilitate the instrumentation efforts. Third, since programming is an expensive activity, it is very desirable that a target problem should allow convenient generation of multiple versions of a program.

5.1 The 3-version MESS Program Experiment

MESS (Mini-Text Editing System) was a program assignment for the graduate seminar course E226Z, offered at UCLA in the Spring quarter of 1976. A preliminary report on MESS is contained in [9]. The specification of MESS and a detailed description of the results can be found in [10].

From the experience and the results of the MESS experiment the following conclusions were reached: (1) The methodology used to implement N-version programming (see Sections 2 and 4) is relatively simple and can be generalized to other similar applications. (2) The results attained from executing 3-version programs are encouraging. The effectiveness of 3-version programming seems to warrant further investigation. (3) The 3-version redundancy was succesfully applied at subroutine (module) level, rather than at complete program level. This shows that selective application of N-version redundancy to certain critical parts of longer programs can be a practical alternative.

5.2 The 3-version RATE Program Experiment

RATE, standing for Region Approximation and Temperature Estimation, is a program for computing dynamic changes of temperatures at discrete points in a particular region of a plain. The temperature changes are governed by the following equation:

$$A \frac{\partial^2 \phi}{\partial x^2} + B \frac{\partial^2 \phi}{\partial y^2} + C \frac{\partial \phi}{\partial x} + D \frac{\partial \phi}{\partial y} = E \frac{\partial \phi}{\partial t} + F$$

where the coefficients A, B, C, D, E, and F are some constants.

The problem specification for RATE (given in [10]) states the requirements for a stand-alone RATE program This specification is written to make a RATE program easily instrumentable for the purpose of conducting experiments on 3-version programming.

There are four types of output data structures described in the problem specification for RATE. The MESSAGE type of data informs the driver about execution termination conditions or the time at which results are produced. The FACTOR type of data represents the size and the units of the specified region. The GRID type of data represents a point in the concerned region and its previous and current temperatures. The OUTPUT-REQUEST type of data indicates the points and the times for which outputs are requested. Each of these types of data can be treated as a c-vector that represents results to be voted upon.

The problem specification for RATE also gives an algorithm specification for: (1) terminating conditions for program execution; (2) cc-points at which results should be produced; and (3) an algorithm that implements a numerical approximation for solving the partial differential equation shown above. Three algorithms, (ALG1, ALG2, ALG3) were selected that implemented three different numerical methods to solve the partial differential equation. While an individual RATE specification can employ only one of these algorithms. the existence of three different choices provides a higher probability of avoiding related programming errors in the N-version programming experiment.

The problem of RATE was given as a programming assignment for the graduate seminar course, E226Z, offered at UCLA in the Spring quarter of 1977. The students were asked to form teams of two people to solve the RATE problem. There were three students who preferred to work alone and their requests were granted. Totally, there were 18 teams thus formed. Two of these teams failed to turn in their programs. That left 16 programs available for this investigation. The RATE specification was distributed to all 18 programming teams. Algorithms 1, 2, and 3 were specified in six cases. Each of the RATE programs was written in P1/I(F), and executed on the IBM 360/91. Each program consisted of more than 600 P1/I statements. The period allowed for the development of the RATE programs was four weeks.

The RATE programs were collected and tested against six text cases. Based on the results of these tests, the best four programs were selected for further experiments. Besides, there were three programs developed by the authors. Hence, we had 7 programs available for subsequent studies on the feasibility of 3-version programming.

Three of the selected programs implemented ALG1. Two each of the remaining four programs implemmented ALG2 and ALG3 respectively. The seven programs were instrumented, grouped into 12 combinations, and tested with 32 test cases. Totally, there were 384 cases tested. Among them:

(1) 290 cases contained no bad version,
(2) 71 cases contained one bad version,
(3) 18 cases contained two bad versions, and
(4) 5 cases contained three bad versions.

Naturally, the 290 cases of three good versions generated acceptable results, and the 18 cases containing two bad versions and the 5 cases containing three bad versions generated unacceptable results. For the 71 cases containing a single bad version, 59 cases

generated acceptable results and 12 cases generated unacceptable results. These 12 cases contained a version which malfunctioned in a way to cause the system to abort the execution of the involved 3-version unit.

Based on these results, we have observed two difficulties which require special attention in N-version programming: (1) In somme situations, one version of code developed an error that caused the operating system of the IBM 360/91 computer to take over execution. An example is the improper handling of conversion errors by a version. As a result, both the involved version and its associated 3-version program were aborted by the operating system. Even though the other two versions were executing properly, the 3-version program could not proceed further past this point to generate correct results. (2) The logic being implemented by one version of the code may be correct, incorrect, or it may be altogether missing. For cases in which missing logic is the cause of incorrect software operation, error symptoms from faulty versions tend to be the same. There is a possibility that faulty but identical results (due to missing logic) may outvote correct results.

6. Conclusions

The results obtained from the MESS and the RATE experiments are of mixed nature. There are several encouraging points: (1) The methodology for implementing N-version programming is relatively simple and can be generalized to other similar applications; (2) In some cases, 3-version programming has been effective in preventing failure due to defects localized in one version of code; and (3) N-version programming can be a practical approach if it is selectively applied at subroutine level. On the other hand, there are some negative points: (1) In the environment of some operating systems, certain implementation defects of a version may cause its associated 3-version program to be aborted by the O.S.; and (2) If missing program functions are the predominant software defects, then N-version programming may not be an effective approach.

In addition to the experimental results, we have identified some situations in which N-version programming appears to be not effective or is not applicable. The most significant limitations are discussed below.

(1) In a real-time environment a system failure may be caused by performance limitations rather than functional problems. Two typical examples are timing constraint violations and resource contentions. A likely source for these problems is system overload. N-version programming may produce adverse effects in these situations.

(2) In certain other circumstances, there exists no unique path to the solution of a problem. Step-by-step matching or voting of correspondent results cannot be used as a criterion of correctness. Therefore, N-version programming is not applicable for situations in which distinct multiple solutions (or intermediate solutions) exist.

(3) In still other cases a long sequence of outputs may not lend itself to be specified in a specific order. In these cases, the outputs from the component versions cannot be readily compared.

(4) In some situations the sequence of outputs from a version is context-dependent. Any error that pushes the rest of output off its proper position makes the subsequent comparison of results meaningless.

(5) In the event that an allowable range of discrepancy cannot be easily determined, the problem of inexact voting is difficult to handle. Acceptable results are difficult to reach in this case.

Although the above conclusions have a significant number of negative points, it should be noted that only a very small portion of the field of N-version programming has been touched. Some negative results might be due to inexperience of programmers, inadequate selection of problems, or improper control of environment for conducting these studies. Furthermore, there are several variations of N-version programming. This approach might be more effective when it is applied to the specification or design of a program [17]. It is also possible that N-version programming can aid program testing more effectively than it can perform run-time software defect masking. Therefore, it is believed that at the present stage of the investigation N-version programming remains an interesting and potentially effective approach to software fault-tolerance.

7. Acknowledgment

This research was supported by the U.S. National Science Foundation, Grant No. MCS 72-03633 A05. The authors wish to acknowledge the generous and enthusiastic advice and cooperation received from Prof. Daniel M. Berry and the programs contributed by the students of his Software Engineering classes in 1976 and in 1977. Valuable advice and criticism was received from Professors J.A. Goguen, Jr. and D.F. Martin of UCLA, and from two referees of this paper.

8. References

[1] A. Avizienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," Proc. 1975 Int. Conf. Reliable Software, 458-464.

[2] B. Randell, "System Structure for Software Fault-Tolerance," IEEE Trans. Software Engr., Vol. SE-1, June 1975, 220-232.

[3] H. Hecht, "Fault-Tolerant Software for Real-Time Applications," ACM Computing Surveys, Vol. 8, Dec. 1976, 391-407.

[4] A. Avizienis, "Fault-Tolerant Computing: Progress, Problems and Prospects," Information Processing 77 (Proc. IFIP Congress 1977), 405-420.

[5] W.R. Elmendorf, "Fault-Tolerant Programming," Proc. 1972 Int. Symp. Fault-Tolerant Computing, June 1972, 79-83.

[6] H. Kopetz, "Software Redundancy in Real Time Systems," Proc. IFIP Congress 1974, 182-186.

[7] E. Girard and J.C. Rault, "A Programming Technnique for Software Reliability," Proc. 1973 IEEE Symp. on Computer Software Reliability, 44-50.

[8] M.A. Fischler, et. al., "Distinct Software: An Approach to Reliable Computing," Proc. 2nd USA-Japan Computer Conference, Tokyo, Japan, 1975, 1-7.

[9] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Execution," Proceedings of COMPSAC 77, (First IEEE-CS International Computer Software and Application Conference), Nov. 1977, 149-155

[10] L. Chen, "Improving Software Reliability by N-version Programming," UCLA Computer Science Dept. Technical Report, University of California Los Angeles, 1978.

[11] B.H. Liskov and V. Berzins, "An Appraisal of Program Specifications," Computation Structures Group Memo 141-1, MIT Laboratory for Computer Science, April 1977.

[12] W.G. Bouricius, et. al., "Reliability Modeling Techniques for Self-Repairing Computer Systems," Proc. ACM 1969 Annual Conf., 295-309.

[13] J.H. Wensley et. al., "The Design, Analysis, and Verification of the SIFT Fault-Tolerant System," Proc. 2nd Int. Conf. Software Engineering, Oct. 1976, 258-269.

[14] A.L. Hopkins, Jr., and T.B. Smith III, "The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor," IEEE Trans. Computers, Vol. C-24, May 1975, 498-505.

[15] J.R. Sklaroff, "Redundancy Management Technique for Space Shuttle Computers," IBM J. of Res. and Dev., Vol. 20, Jan. 1976, 20-28.

[16] R.B. Broen, "New Voters for Redundant Systems," Trans. ASME: Journal of Dynamic Systems, Measurement, and Control, March 1975, 41-45.

[17] A.B. Long, C.V. Ramamoorthy, et. al., "A Methodology for Development and Validation of Critical Software for Nuclear Power Plants," Proc. COMPSAC 77 (IEEE-CS Int. Computer Software & Applications Conf.), 620-626.

Authors

Liming Chen was born in Taiwan. He received the B.S. ('69) in Psychology from the National Taiwan University and the M.A. and M.S. degrees in Psychology and in Computer Science from UCLA. Since 1974 he has been a Postgraduate Research Engineer associated with the Fault-Tolerant Computing project at UCLA, where he is completing the Ph.D. dissertation in Computer Science. Since May 1977, he is associated with the Xerox Co. working on software development methodology, diagnostic programming, and human engineering.

Algirdas Avižienis was born in Kaunas, Lithuania. He received the B.S. ('54), M.S. ('55) and Ph.D. ('60) degrees in Electrical Engineering from the University of Illinois. In 1960 he initiated research on fault-tolerant computing and later directed the JPL-STAR computer project at the Jet Propulsion Laboratory, where he is now an Academic Member of Technical Staff. Since 1962 he has been a faculty member at UCLA, where he is now a Professor in the Computer Science Department. He is a Fellow of the IEEE and the author of over 50 publications on computer arithmetic and fault tolerance. He was the first chairman of the IEEE-CS Technical Committee on Fault-Tolerant Computing and the Chairman of FTCS-1 in 1971.

Appendix

Assume that a 3-version software unit has been identified to implement a function which can be activated repeatedly to convert an array of 10 ASCII coded digits into a binary number. Possible implementation of 3 versions of a program for this function are shown in Figures 5, 6, and 7. It is evident that these implementations are simple derivations of that in Figure 2.

```
CONVERSION1: PROCEDURE OPTIONS (TASK);
   DCL NUMBER1     BINARY     FIXED(31) EXTERNAL;
   DCL (SERVICE1, COMPLETE1) EVENT      EXTERNAL;
   DCL (DISAGREE1, GOODBYE)  BIT(1)     EXTERNAL;
   DCL DIGITS(10)  BINARY     FIXED(6)  EXTERNAL;
   DCL FINIS       BIT(1)     INIT('0'B);
   DO WHILE (¬FINIS);
      WAIT (SERVICE1);
      COMPLETION (SERVICE1) = '0'B;
      NUMBER1 = 0;
      IF ¬GOODBYE & ¬DISAGREE1
      THEN DO I = 1 TO 10;
              NUMBER1 = NUMBER1*10+
                        DIGITS(I)-60;
           END;
      ELSE FINIS = '1'B;
      COMPLETION (COMPLETE1) = '1'B;
   END;
   END CONVERSION1;
```

Figure 5

```
CONVERSION2: PROCEDURE OPTIONS (TASK);
   DCL NUMBER 2    BINARY     FIXED(31) EXTERNAL;
   DCL (DISAGREE2, GOODBYE)  BIT(1)     EXTERNAL;
   DCL (SERVICE2, COMPLETE2) EVENT      EXTERNAL;
   DCL (DIGITS(10) BINARY     FIXED(6)  EXTERNAL;
   DCL FINIS       BIT(1)     INIT('0'B);
   DO WHILE (¬FINIS);
      WAIT (SERVICE2);
      COMPLETION (SERVICE2) = '0'B;
      NUMBER2 = 0;
      IF ¬GOODBYE & ¬DISAGREE2
      THEN DO I = 1 TO 10;
           NUMBER2 = NUMBER2 +
                     (DIGITS(I)-60)*10**(10-I);
           END;
      ELSE FINIS = '1'B;
      COMPLEION (COMPLETE2) = '1'B;
   END;
   END CONVERSION2;
```

Figure 6

```
CONVERSION3: PROCEDURE OPTIONS (MAIN);
   DCL NUMBER3     BINARY     FIXED(31) EXTERNAL;
   DCL (DISAGREE3, GOODBYE)  BIT(1)     EXTERNAL;
   DCL (SERVICE3, COMPLETE3) EVENT      EXTERNAL;
   DCL DIGITS(10)  BINARY     FIXED(6)  EXTERNAL;
   DCL FINIS       BIT(1)     INIT('0'B);
   DO WHILE (¬FINIS);
      WAIT (SERVICE3);
      COMPLETION (SERVICE3) = '0'B;
      NUMBER3 = 0;
      IF ¬GOODBYE & ¬DISAGREE3
      THEN DO I = 1 TO 10;
           NUMBER3 = NUMBER3*10+
                     MOD(DIGITS(I), 60);
           END;
      ELSE FINIS = '1'B;
      COMPLETION(COMPLETE3) = '1'B;
   END;
   END CONVERSION3;
```

Figure 7