SOFTWARE FAULT-TOLERANCE BY DESIGN DIVERSITY DEDIX: A TOOL FOR EXPERIMENTS

A. Avižienis, P. Gunningberg[1], J. P. J. Kelly, R. T. Lyu, L. Strigini[2], P. J. Traverse[3], K. S. Tso, U. Voges[4]

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

Abstract. A large number of computing systems require very high levels of reliability, availability, or safety. A fault-avoidance approach is not practical in many cases, and is costly and difficult for software, if not impossible. One way of reducing the effects of an error introduced during the design of a program is to use multiple versions of the program, independently designed from a common specification. If these versions are designed by independent programming teams, it is to be expected that a fault in one version will not have the same behavior as any fault in the other versions. Since the errors in the output of the versions will be different and uncorrelated, it is possible to run the versions concurrently, cross-check their results at prespecified points, and mask errors. A DEsign DIversity eXperiments (DEDIX) testbed has been implemented at UCLA to study the influence of common mode errors which can result in a failure of the entire system. The layered design of DEDIX and its decision algorithm are described. The usage of the system and its application in an ongoing experiment are explained.

Key words. Computer Architecture, Reliability Theory, Distributed Parameters Systems, Coding Errors, Fault Tolerance.

INTRODUCTION

A large number of contemporary computing systems intended for process control applications have stringent reliability and availability requirements. This means that they must deliver the output in a timely manner with a high probability of being correct. Such process control computers with high *dependability* goals can be found, for example, in the nuclear and aerospace industries. A simple and efficient way of reaching this dependability goal is to use an *error masking* approach. An error can be masked if the system is provided with enough redundancy: typically, the execution of multiple (N-fold) computations, each computation having the same objective [Avižienis1984]. The output of each computation then is voted on by a more or less sophisticated decision algorithm. The result is either a single output or one output for each computation channel which is within a specified, acceptable tolerance.

In order to allow dependable voting on the output, only a minority of the computation channels may produce an error at a given decision point. This condition is one of the basic assumptions needed for successful voting. Furthermore, if

- the inputs to each computation channel are consistent,
- the outputs are voted upon (in a more or less sophisticated decision function), and
- the probability of having *related errors* is sufficiently low,

then, the output of the system is sufficiently dependable.

These assumptions are usually satisfied. The most troublesome deals with related errors. This assumption is very important, because, if one error appears simultaneously in a majority of channels, any decision function will produce an incorrect result. Therefore, this probability of common mode error has to be kept low.

As long as certain design criteria are obeyed, these related errors are not likely to appear if they are due to internal physical faults (rupture of connection,e. g.), as these faults are likely to have an effect only on one of the channels at a time. External faults are more likely to produce related errors. Ways of dealing with these errors are to have the channels loosely coupled, and to use different technologies for the channels. Then, an external fault will not strike the channels when they are in the same state, and they will not react in the same way. They are thus *distinguishable*.

Another source of related errors are design errors. Indeed, the N copies of faulty software will all be in error at the same time when provided with identical input data. A way to avoid these related errors is to have different versions of the software (and of the entire channels) instead of using simple copies. Thus a key attribute for high dependability systems appears to be *diversity*: diversity in the timing, technology, and design (hardware and software) of the different channels.

Let us define a *cross-check point* (cc-point): to be the voting point at which the different versions exchange their results (cc-vector) for voting. The basic assumption, that only a minority is in error, can then also be expressed as: between two successive cc-points only a minority of the redundant channels are likely to fail, either by procuding erroneous output or by failing to deliver their result in time. Errors in the computation will have an effect on

[1] *On leave from Uppsala University, Sweden*
[2] *On leave from IEI-CNR, Pisa, Italy*
[3] *On leave from LAAS, Toulouse, France*
[4] *On leave from KFK, Karlsruhe, F.R. Germany*

this cc-vector and are therefore detectable. The decision algorithm will compare the cc-vectors and will output its result in form of a decision vector.

At UCLA an ongoing research effort was started to investigate design diversity, the problems that can arise, and to estimate the efficiency in dependability improvement by the use of design diversity. The main target is the software, and first results included the definition of the concept of *N-Version Programming*[Chen1978], and some first generation experiments [Kelly1982].

In order to make measurements in a multi version software experiment, a testbed was needed. A basic requirement was to simulate the environments in which design diversity should be used. The Design Diversity Experiments testbed (DEDIX) has thus two aspects: a fault-tolerant computing system, and an experimentation tool. We will develop these two aspects in this paper. The main layout of the DEDIX system will be given and the decision algorithm implemented in DEDIX will be explained more closely. Finally, the use of DEDIX in current experiments will be described. A more complete description of DEDIX can be found in [Avižienis1985].

DEDIX AS A FAULT-TOLERANT COMPUTING SYSTEM

As stated earlier, design diversity will often be used in an environment with high redundancy. Therefore, the testbed has to be a modular, redundant system to allow different experiments. the basic requirements for DEDIX are the following:

1. The different versions of the software shall be able to run on different hardware in order to test the influence of errors in the hardware associated with any one version. Version support software, therefore, has to be distributed.
2. DEDIX must run on the distributed Locus environment at UCLA [Walker1983], consisting of a network of about 20 VAX 11/750s, and should be portable to other Unix systems.
3. A decision algorithm has to be part of the system, which provides different kinds of decision functions for the user like bit-by-bit comparison for identity, and comparison within a specified tolerance.
4. The interface for the version programmer has to be simple, and the interface must be independent of the number of actual versions used.

In order to fulfil these requirements, DEDIX was developed as a modular redundant system. Depending on the number of versions and the number of available machines, DEDIX selects appropriate hardware.

DEDIX itself is written in C and makes use of several Locus features, e. g. for setting up the different processes and for linking the processes via pipes. Nevertheless, it should be possible to port DEDIX to a pure Unix system which provides mechanisms for communication between several CPUs.

We use the facilities offered by the UCLA Center for Experimental Computer Science. The machines are linked by an Ethernet local network. We use the Unix software development environment and its inter-process communication features (pipes). Locus allows processes to communicate with each other in the same way whether they are running on the same machine or on different machines. It is thus easy to allocate each computation channel to a different machine.

The decision algorithm implemented will be described in more detail later, as well as the user interface. Both parts are designed to fulfil the above mentioned requirements.

A global view of the DEDIX system supporting N versions is given in Fig. 1. The versions communicate with the different parts of DEDIX, which in turn makes use of the Locus operating system, and the different sites are interconnected with each other via Ethernet.

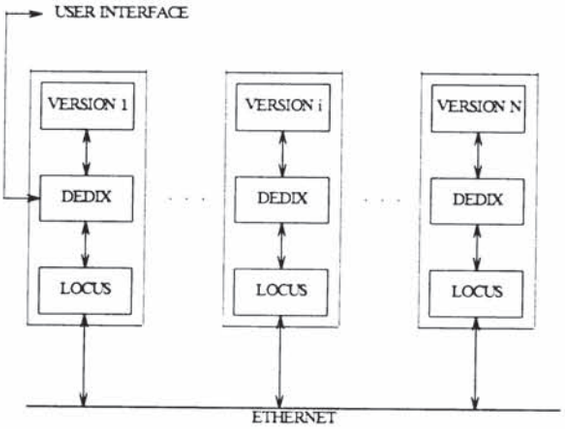

Fig. 1. The N sites of DEDIX

DEDIX: A LAYERED APPROACH

DEDIX is designed as a set of hierarchically structured layers. Each of the sites which are selected for running DEDIX has an identical set of layers and entities, providing services to its version and the external user. These layers, from top to bottom, are:

- the Version Layer,
- the Decision and Executive Layer,
- the Synchronization Layer,
- the Transport Layer.

These layers are implemented as functions, and inside a site, they share some data structures (see Fig. 2).

The Version Layer

This layer contains the application program version. The purpose of this layer is to interface the version with the DEDIX system. The interface function is called the cross-check, or *cc-function* since it is called by the version at each cc-point. Pointers to the results to be corrected are sent as parameters to this function. The cc-function transfers the version representation of results into a cc-vector so that the DEDIX internal representation of a cc-vector is hidden for the version program. If the decision algorithm detects an error in the results of the version, the cc-function writes back the corrected results into the version, therefore masking errors.

To run on DEDIX a version must be instrumented. That is, the version must call DEDIX at each occurrence of a

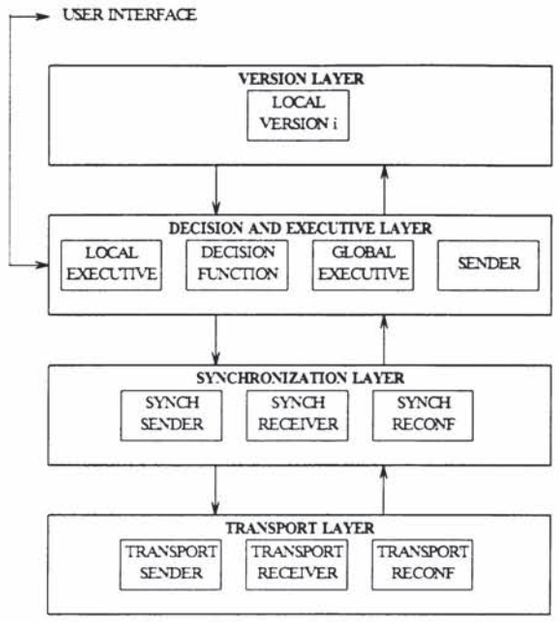

Fig. 2. The layers on one site of DEDIX.

cc-point, and pass its results to generate the corresponding cc-vectors. We will show how this is done in a subsequent section. Currently, the available application languages are C and Pascal. Other languages could be used for the versions, if the interface between this language and C is provided.

The Decision and Executive Layer

This layer receives cc-vectors from the versions, decides on the correct result, determines whether a version is faulty or not, and makes recovery decisions. A corrected cc-vector is forwarded to the version. All exceptions that cannot be handled at lower levels are directed to this layer.

The layer has four entities, a *sender*, a *local executive*, a *decision function*, and a *global executive*. The local executive entity receives requests from the version and responds to the version when a decision has been taken. There are four different types of normal requests: intermediate cc-vector (on a subset of the internal state of the channels), output cc-vector, input, and version termination. All of them are broadcast to the other sites, and run through the decision function to ensure consistency and synchronization. When the version has raised an exception from which it cannot recover, this exception is forwarded to the local executive.

The global executive is activated when the decision function indicates that the result is not unanimous, or when some unrecoverable exception is signaled from the version or some other layer. Such an exception could be disruption of a communication connection. This global executive provides fault diagnosis, reconfiguration, and fault reporting for maintenance purposes. Basically, it has the same functions as the global executive found in SIFT [Melliar-Smith1982].

To ensure that a consistent reconfiguration decision is taken, the global executive at each site must first get a consistent error report. All global executives propose a new configuration that is broadcast to every site and decided upon. The proposed configurations are voted on bit-by-bit which will ensure a consistent view on a new configuration at every correctly working site.

The Synchronization Layer

For each physically distinct site, this layer broadcasts the result from the above executive layer and collects messages with the results ("cc-vector") from all other sites. This layer only accepts messages that are both broadcast within a certain time interval and that will arrive within the same time interval. The collected messages are delivered to the decision function. A new set of results is accepted when every site has confirmed that the messages have been delivered. This layer can establish communication connections between sites.

A protocol was designed to provide the above service. Synchronization of the system is based on the following assumptions:

- correctly working versions produce exactly the same number of cc-vectors,
- correctly working versions have similar execution times, i.e. they will produce results within a specified time-out interval,
- a majority of "missing" messages does not exist at a majority of sites,
- a majority of messages are not delayed more than the specified time-out interval.

Each site has both a sender and a receiver entity in this layer, which communicate with corresponding entities of other sites according to the protocol. The receiver entity collects messages from the senders and it delivers them to the decision function. After the delivery, it sends acknowledgments back to the senders to confirm the delivery. When a sender entity has collected acknowledgements from all the other sites or when it has at least a majority of acknowledgments, it will indicate this to its decision and executive layer. This indication is used by the layer above to restart the version. By using this indication, it is possible to ensure that all sites will start the new set of computations within the specified time interval.

The senders and receivers are designed as communicating extended finite state machines. They respond to events such as commands from the local executive, messages or acknowledgments, and internal time-outs. State variables, i.e. frame sequence numbers, forming predicates on the state transitions are used to discriminate messages and acknowledgments delayed too long in the communication system. The specification and verification of the protocol is described in [Gunningberg1985].

The Transport Layer

This layer controls the communication of messages (containing the results) between the sites. Messages are broadcast to all active sites. The layer makes sure that no message is lost, duplicated, damaged, or misaddressed, and it preserves the ordering of sent messages. A disconnection is reported to the layer above.

Currently, this layer is implemented as a simple loop of

point to point links by UNIX interprocessor pipes. Since this implementation does not allow a site crash, a redundant interconnection structure is under implementation. We are also investigating the use of network-oriented inter-process communication protocol to achieve more transportation efficiency [Cooper1984].

THE DECISION FUNCTION OF DEDIX

The decision function has to recognize whether the versions are in agreement with each other or not. The decision function is used for each cc-point, and each of these decisions is independent of the preceding ones, and based only on the set of cc-vectors that is transmitted by the synchronization layer. An agreement is achieved if at least a majority of versions is considered to be equivalent by the decision algorithm, and this value is used as an output. This value is also communicated to the versions in error, so they can use it for their subsequent computation.

An agreement among cc-vectors means basically that these cc-vectors contain the same information, at the level of abstraction of the user of the versions. This means that the versions (that have been designed by different programmer teams, in different languages, that may run on different machines, ...) may have different ways of representing information. The decision function has thus to extract the meaning of the cc-vectors. A "bit-by-bit" vote can be used for much of the cc-vector since there is only one possible representation of the data. Nevertheless previous experiments have shown that bit-by-bit voting can be too selective and reject semantically equivalent results [Kelly1982].

Therefore, the cc-vectors is subdivided into parts, and a separate decision is possible for each part. The global decision vector is composed of the union of the values of each part. The parts can be classified in the following way:

- "matching class", where a bit-by-bit vote is used (primarily for integers),
- "cosmetic class", where cosmetic errors are allowed (mainly used for character strings),
- "real number class", containing real numbers which are allowed to be slightly different.

Each class is considered separately below.

Matching Decision

This decision is applied on data that must be strictly equal, like binary values or integers. The comparison on equality is done between all cc-vectors.

Cosmetic Decision

Cosmetic errors are defined as errors in character strings like minor misspelling in a word which is to be displayed to the operator. The human would recognise the error and still correctly understand the word or message. If diverse versions are used with a bit-by-bit vote, a "cosmetically faulty" version will be declared faulty, and, according to the reconfiguration policy, could be discarded. If, on the other hand, the decision function can tolerate cosmetic errors, a system using design diversity will not be penalized in comparison to a "classical" fault-tolerant system. A version with cosmetic errors need not be discarded. However a cosmetic error must be distinguished from a fatal error.

As an example consider the integer '9', it can be written as character string '09', '9', or '_9', which would result in disagreement in a bit-by-bit comparison. In contrast, if the word size and the number representation are defined, the comparison of '9' as an integer would result in only one possible representation. Therefore numbers should not be represented as character strings.

For character strings, we have to decide which misspellings to allow. In a study [Pollock1983] misspellings found in several journals have been categorised. As the text of these journals has been processed by computer, the kind of misspellings in them can be expected to be representative of faults entered through a keyboard, and so representative of software. The Study showed that one misspelling occured for every 250 words. More than 90% of these misspellings can be characterized as being

- an omission of one character,
- an insertion of one character,
- a substitution of one character by another one,
- a transposition of two adjacent characters.

Cosmetic errors are tolerated by the cosmetic decision if they are part of the above four cases.

Numeric Decision

For decisions on real numbers, two solutions are proposed: select one representative value or tolerate all values within a given tolerance. In the first case, the representative value has to be defined and its selection algorithm has to be implemented, which will always result in an acceptable solution. In the second case, the results of the different versions are compared with each other to determine whether a majority of them is close enough together within the tolerance. Currently, the first approach is implemented in DEDIX, since we have been able to derive a very simple decision algorithm. This algorithm is summarized in the following.

We assume that an ideal value exists (IDEAL_VALUE), from which an allowed imprecision is defined (δ_-,δ_+), such that a version V_i is assumed to be non faulty, if and only if its response (R_i) is such that:

$$\text{IDEAL\_VALUE} - \delta_- \leq R_i \leq \text{IDEAL\_VALUE} + \delta_+.$$

The key of the algorithm is that it can be proved that, so long as a majority of versions are not faulty, the median of all responses is such that:

$$\text{IDEAL\_VALUE} - \delta_- \leq \text{MEDIAN} \leq \text{IDEAL\_VALUE} + \delta_+.$$

Since taking the median of numbers is very easy to do, we have thus a very simple way to compute a decision value. The most diverging versions can also be detected, as, under the same condition as the preceding property, it can be proved that a version V_i is faulty if

$$\text{MEDIAN} + \delta_- + \delta_+ < R_i$$

or

$$R_i < \text{MEDIAN} - \delta_- - \delta_+.$$

The agreement is reached in the following steps:

- computation of the median of the skews (if the versions use different skews),
- computation of the median of the responses,
- filtration of the versions using the above medians.

An agreement exists if a majority of versions has not been discarded by the filter; the decision value is the median.

DEDIX AS AN EXPERIMENTATION TOOL

Program Interface

In multiple version software the versions of an application program are all written according to the same functional specification. The specification must dictate not only the overall input-output transformation the program has to perform, but also which intermediate results must be compared, and at which points in the execution. The difference between a non-redundant program and the corresponding multiple version software running on DEDIX is minimized for programmers. Figure 3 shows a program written in C and its corresponding instrumented version. The program continues to read the system clock and output the current time until the user stops it.

```
main () {
    char *ctime();
    long clock;
    double f_clock;
    char *ctime_ret;
    char *reply = "y\n                                    ";
    static char *s1 = "\tDate is: ";
    static char *s2 = " Do we continue? (y/n) ";
    while (reply[0] == 'Y' || reply[0] == 'y') {
        f_clock = time(0);
        clock = f_clock;
        ctime_ret = ctime(&clock);
        printf ("%s%s%s", s1, ctime_ret, s2);
        scanf ("%s", reply);
    }
    exit (0);
}
```

(a) basic program

```
version () {
    char *ctime();
    long clock;
    double f_clock;
    float f_drift = 2.0;
    char *ctime_ret;
    char *reply = "y\n                                    ";
    static char *s1 = "\tDate is: ";
    static char *s2 = " Do we continue? (y/n) ";
    while (reply[0] == 'Y' || reply[0] == 'y') {
        f_clock = time(0);
        ccpoint(1, "%k%e", &f_drift, &f_clock);
        clock = f_clock;
        ctime_ret = ctime(&clock);
        ccoutput(2, "%S%s%S", s1, ctime_ret, s2);
        ccinput(3, "%s", reply);
    }
    return (0);
}
```

(b) instrumented version

Fig. 3. A program for displaying current time.

The differences between the program and the version are as follows:

(1) The name of the main function of the program is changed from main () to version ().

(2) The Cross-check function is called to decide on the clock values of different versions after the system clock is read. The first argument specifies the cc-point id. The second is the format which specifies that the clock value is voted on as a real number with a specified skew.

(3) Instead of using printf function for standard output, the ccoutput function is used which first votes on the output values and then outputs them. %S specifies that the string can tolerate cosmetic error.

(4) Similarly for the input, ccinput is used to input data from the standard input and broadcast it to all the versions.

(5) At the end of the program, *return* is used instead of *exit*.

User Interface

The user interface of DEDIX allows users to debug the system as well as the versions, monitor the operations of the system, apply stimuli to the system, and to collect empirical data during experimentation. A number of commands are available to the user for controlling the execution and defining additional output.

Breakpoint. The *break* command enables the user to set breakpoints. At a breakpoint, DEDIX stops executing and goes into the user interface where the user can enter commands to examine the current system states, examine past execution history, or inject stimuli to the system.

Monitoring. The user can examine the current contents of the message passing through the transport layer by using the *display* command. Since every message is logged, the user may also specify conditions in the *display* command to examine any message logged in the past. The user can also examine the internal system states by using the *show* command, e.g., to examine the breakpoints which have been set, the results of voting, etc.

Stimuli Injection. The user is allowed to inject faults to the system by changing the system states, e.g., the cc-vector, by using the *modify* command.

Statistics Collection. The user interface gathers empirical data and collects statistics of the experiments. Every message passing the transport layer is logged into a file with a time-stamp. This enables the user to do post-execution analysis or even replay the experiment. Statistics like elapsed time, system time, number of cc-points executed, and their results of decision are also collected.

Experiments

Several systems are already using diverse software, e.g. [Anderson1985, Gmeiner1979, Martin1982, Taylor1981]. Nevertheless, it appears (in addition to the fact that some people are not yet convinced of the usefulness of design diversity) that we need to know more about related errors. A primary goal of DEDIX is thus to evaluate these related errors. By using a controlled environment, it will be possible to examine the errors in order to

- trace the related errors,
- know whether the proportion of related errors is important or not,
- know the impact they have on the dependability of the system.

The data so obtained will be used to evaluate the meaning of design diversity and the architecture of future fault-tolerant computers.

Another important goal of DEDIX is the evaluation of specification methods. Indeed, specifications are likely to be the "hard-core" and the choice of a specification method has thus to be carefully evaluated. The number and proportion of related errors is a measure of the efficiency of a specification method. By efficiency, we mean the inherent ability of the method to reduce errors and other ambiguities in the resulting specifications.

What about the cost? It has been claimed that design diversity was too costly to be used. This is obviously not the case when the cost of a failure of the system is important (money or lifes). Without claiming as [Gilb1974] that N-version programming will always reduce programming cost, we consider the advantage of testing the versions in parallel, with DEDIX for example. Indeed, the test data are applied to all versions together, and no reference is needed: the reference is given by the agreeing majority of the versions.

To avoid effecting the execution time of DEDIX, the experimentation analysis is performed off-line. During the execution, files are created with for each occurrence of a cc-point, the cc-vectors of all the versions, the decision vector, and the diverse diagnosis and reconfiguration decision available in DEDIX.

CONCLUSION

Currently, DEDIX is completely implemented and running. The initial number of versions can be 2 or more, and a graceful degradation occurs when a version is rejected as being too often faulty. An experiment is under design, under the management of NASA, with the collaboration of four universities (University of Virginia, University of Illinois, North Carolina State University, and UCLA). After these experiments, some other fault-tolerance techniques will be tried on DEDIX (particularly in the domain of reconfiguration and recovery).

ACKNOWLEDGMENT

The research described in this paper has been supported by the Advanced Computer Science program of the FAA, by NASA contract NAG1-512, and by NSF grant MCS 81-21696.
We thank Jean-Claude Laprie for discussing this paper with us and giving as some valuable remarks.

REFERENCES

Anderson, T., Barrett, P. A., Halliwell, D. N., and Moulding, M. R., "An Evaluation of Software Fault Tolerance in a Practical System," in *Proceedings 15th Internat. Symp. on Fault-Tolerant Computing*, Ann Arbor, MI: 19-21 June 1985.

Avižienis, A. and Kelly, J., "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol. 17, No. 8, August 1984, pp. 67-80.

Avižienis, A., Gunningberg, P., Kelly, J.P.J., Strigini, L., Traverse, P.J., Tso, K.S., and Voges, U., "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software.," in *15th IEEE International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan: June 1985.

Chen, L. and Avižienis, A., "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Proceedings 8th IEEE International Symposium on Fault-Tolerant Computing Systems*, Toulouse, France: June 1978, pp. 3-9.

Cooper, E.C., "A replicated Procedure Call Facility," in *Proceedings 4th Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, MD: October 1984.

Gilb, T., "Parallel Programming," *Datamation*, October 1974, pp. 160-161.

Gmeiner, L. and Voges, U., "Software Diversity in Reactor Protection Systems: An Experiment," in *Proceedings Safety of Computer Control Systems, IFAC Workshop*, Stuttgart, Federal Republic of Germany: May 1979, pp. 73-79.

Gunningberg, P. and Pehrson, B., "Protocol and Verification of a Synchronization Protocol for Comparison of Results.," in *15th IEEE International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan: June 1985.

Kelly, J.P.J., "Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach," UCLA, Computer Science Department, Los Angeles, California, Tech. Rep. CSD-820927, September 1982.

Martin, D.J., "Dissimilar Software in High Integrity Application in Flight Controls," in *Proceedings AGARD-CPP-330*, September 1982, pp. 36.1-36.13.

Melliar-Smith, P.M. and Schwartz, R.L., "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 616-630.

Pollock, J.J. and Zamora, A., "Collection and Characterization of Spelling Errors in Scientific and Scholarly Text," *Journal of the American Society for Information Science*, Vol. 34, No. 1, January, 1983, pp. 51-58.

Taylor, R., "Redundant Programming in Europe," *ACM Sigsoft Sen.*, Vol. 6, No. 1, January 1981.

Walker, B.J., Popek, G.J., English, R., Kline, C., and Thiel, G., "The LOCUS Distributed Operating System," in *Proceedings 9th ACM Symposium on Operating System Principles*, Bretton Woods, NH: October 1983, pp. 49-70.