DEDIX 87 - A Supervisory System for Design Diversity Experiments at UCLA

Algirdas Avižienis, Michael R. T. Lyu, Werner Schütz, Kam-Sing Tso, Udo Voges

Dependable Computing and Fault-Tolerant Systems Laboratory
UCLA Computer Science Department
University of California
Los Angeles, CA 90024, USA

Abstract

To establish a long-term research facility for further experimental investigations of design diversity as a means of achieving fault-tolerant systems, the DEDIX (DEsign DIversity eXperiment) system, a distributed supervisor and testbed for multi-version software, was designed and implemented by researchers at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory. DEDIX is available on the Olympus local network, which utilizes the Locus distributed operating system to operate a set of several VAX 11/750 computers at the UCLA Center for Experimental Computer Science. DEDIX is portable to any machine which runs a Unix operating system. The DEDIX system is described and its applications are discussed in this paper. A review of current research is also presented.

1. Introduction

By early 1970s significant progress had been made in the development of systems that tolerate physical faults that are due to random failures of components or physical interference with the hardware of a system. At that time it became clear that design faults, especially as represented by software "bugs", presented the next challenge to the researchers in fault tolerance. A research effort to attain tolerance of design faults by means of multi-version software was started at UCLA in early 1975. The method was first described as "redundant programming" at the April 1975 International Conference on Reliable Software in Los Angeles [Avižienis 1975], and was renamed as "N-version programming" in the course of the next two years [Avižienis 1977]. The name "Multi-Version Software" (MVS) has also been used. The entire UCLA design diversity research effort through mid-1985 has been summarized in [Avižienis 1985b].

The N-version programming approach to fault tolerant software systems employs functionally equivalent, yet independently developed software components. These components are executed concurrently under a supervisory system that uses a decision algorithm based on consensus to determine final output values. From its beginning in 1975, the fundamental conjecture of the MVS approach at UCLA has been that errors due to residual software faults are very likely to be masked by the correct results produced by the other versions in the system. This conjecture does not assume independence of errors, but rather a low probability of their concurrence and similarity. MVS systems achieve reliability improvements through the use of redundancy and diversity. A "dimension of diversity" is one of the independent variables in the development process of an MVS system. Diversity may be achieved along various dimensions, e.g., specification languages, specification writers, programming languages, programmers, algorithms, data structures, development environments, and testing methods.

The scarcity of previous results and an absence of formal theories on N-version programming in 1975 led to the choice of an experimental approach: 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 were executed as N-version software units in a simulated multiple-hardware system, and the resulting observations were applied to refine the methodology and to build up the concepts on N-version programming. The first detailed assessment of the research approach and a discussion of two sets of experimental results, using 27 and 16 independently written programs from a software engineering class, was published in 1978 [Chen 1978].

This exploratory research demonstrated the practicality of experimental investigation and confirmed the need for high quality software specifications. As a consequence, the first aim of the next phase of UCLA research (1979-82) was the investigation of the relative applicability of three distinct software specification techniques: formal (OBJ), semiformal (PDL), and in

English.

Other aims were to investigate the types and causes of software design faults, to propose improvements to software specification techniques and their use, and to propose future experiments for the investigation of design fault tolerance in software and in hardware [Kelly 1983, Avižienis 1984].

In the course of the experiments at UCLA it became evident that the usual general-purpose campus computing services were poorly suited to support the systematic execution, instrumentation, and observation of N-version fault-tolerant software. In order to provide a long-term research facility for experimental investigations of design diversity as a means of achieving fault-tolerant systems, researchers at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory have designed and implemented the prototype DEDIX (DEsign DIversity eXperiment) system, a distributed supervisor and testbed for multiple-version software [Avižienis 1985a]. DEDIX is supported by the Olympus Net local network at the UCLA Center for Experimental Computer Science which utilizes the UNIX-based Locus distributed operating system to operate a set of VAX 11/750 computers.

The purpose of DEDIX is to supervise and to observe the execution of N diverse versions of an application program functioning as a fault-tolerant N-version software unit. DEDIX also provides a transparent interface to the users, versions, and the input/output system, so that they need not be aware of the existence of multiple versions and recovery algorithms. The prototype DEDIX system has been operational since early 1985. Several modifications have been introduced since then, most of them intended to improve the speed of the execution of N-version software. The first major test of DEDIX has been the experimentation with the set of five programs produced at UCLA for the NASA-sponsored Four-University N-version software project. A complete overview of the structure and of the applications of DEDIX at UCLA is presented in this paper.

1.1 Functional Requirements of DEDIX

The principal functional requirements of DEDIX are as follows:

Distribution: the versions must be able to execute on separate physical sites in order to take advantage of physical isolation between sites, to benefit from parallel execution, and to survive a crash of a minority of sites.

Transparency: the application programmer must not be required to write special software to take care of the multiplicity, and a version must be able to run in a system with any allowed value of N without modifications.

Decision making: a reliable *decision algorithm* that determines a single *consensus result* from the multiple version results must be provided. The algorithm must be able to deal with specified allowable differences in numerical values and with slightly different formats (e.g. misspellings) in human-readable results; additionally, the user must be able to choose between different decision algorithms and even -- with some more effort -- be able to incorporate a special decision algorithm of his own. If a consensus cannot be obtained, an alternate decision must be provided.

Recovery and/or reconfiguration: DEDIX must support recovery attempts for the minority of disagreeing versions. It also must implement reconfiguration decisions that remove failed versions or sites when recovery is not available or does not succeed. When a consensus does not result, an alternate outcome (safe shutdown, or invocation of a backup system) must be implemented.

Environment: DEDIX must run on the distributed Locus environment at UCLA [Popek 1981] and must be easily portable to other UNIX systems. DEDIX must be able to run concurrently with all other normal activities of the local network.

1.2 Related Research

The DEDIX system can be considered as a network-based generalization of SIFT [Wensley 1978] that is able to tolerate both physical and design faults in software and in hardware. Both have similar partitioning, with a local executive and a decision algorithm at each site that processes broadcast results, and a copy of the global executive at each site that takes consistent recovery and reconfiguration decisions by majority vote. DEDIX is extended to allow some diversity in results and in version execution times. SIFT is a frame synchronous system that uses periodically synchronized clocks to predict when results should be available for a decision. This technique does not allow the diversity in execution times and unpredictable delays in communication that can be found in a distributed *N*-version environment, especially when it is shared with other jobs. Instead, a synchronization protocol is used in DEDIX which does not make reference to global time within the system.

Another approach to fault-tolerant software is the *Recovery Block* technique, in which alternate software versions are organized in a manner similar to the dynamic redundancy (standby sparing) technique used in hardware [Anderson 1981]. The objective of the recovery block technique is to perform software design fault detection during runtime by an acceptance test performed on the results of one version, as opposed to comparing results from several versions. If the test fails, an alternate version is executed to implement recovery. Several major research activities related to N-version programming and recovery block techniques have been reported [Anderson 1985, Cristian 1982, Kelly 1986, Kim 1984, Ramamoorthy 1981, Voges 1982].

2. Functional Description of the DEDIX System

2.1 Services and Structure

DEDIX together with the diverse program versions has the ability to tolerate software design and implementation faults. DEDIX and the versions interact with each other and with their environment, i.e., a user, so that together they can be seen as a fault-tolerant *multi-version system.* DEDIX itself is a *supervisor* that does not add any application relevant functions to the system.

The purpose of DEDIX is to supervise and to observe the execution of N diverse versions of an application program functioning as a fault-tolerant N-version software unit. DEDIX also provides a transparent interface to the users, the versions, and the input/output system so that they need not be aware of the existence of multiple versions and recovery algorithms. An abstract view of DEDIX as a multiversion system with N versions is given in Fig. 1. Generally speaking, DEDIX provides the following services:

- it handles communications from the user and distributes them to all active versions;
- it handles requests from the versions to have their results (cc-vectors) processed, and returns consensus results to the versions and to the user;
- it executes decision algorithms and determines consensus results, or invokes alternate decisions if a consensus does not exist;
- it manages the input and output operations for the versions; and
- it makes reconfiguration and recovery decisions about the handling of faulty versions.

Partitioning of DEDIX. The DEDIX system can be located either in a single computer that executes all versions sequentially, or in a multicomputer system running one (or more) versions at each site. If DEDIX is supported by a single computer, it is vulnerable to hardware and software faults that

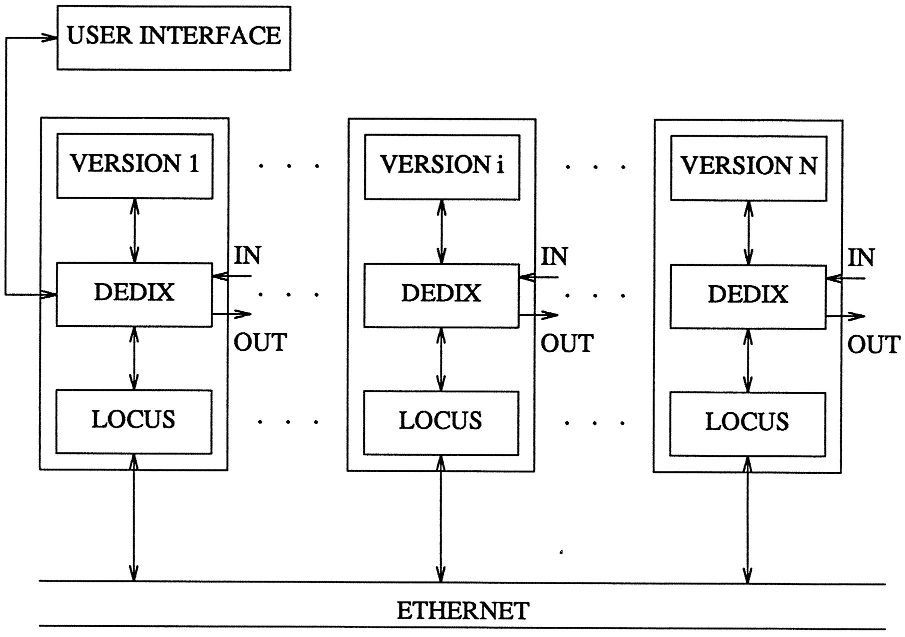

Fig. 1. DEDIX as a Multiversion System

affect the host computer, and the execution of N-version software units is relatively slow. In a computer network environment, the system is partitioned to protect against most hardware faults. This has been done by providing each site with its own local DEDIX software, while an intersite communication facility is common to all computers. The DEDIX design is suitable for any specified number $N \geq 2$ of sites and versions, and currently accommodates the range $2 \leq N \leq 20$, with provisions to reduce the number of sites and to adjust the decision algorithm upon the failure of a version or a site.

The manifestation of faults. A hardware or software fault will affect a program version and it may also affect the underlying system. DEDIX is designed to be able to identify a malfunctioning site and to tolerate both cases of fault effects, provided that the errors can be detected. In the first case, when the errors and the faults can be isolated to a version only, the site will attempt to recover the internal state of the local version with decision results. In the second fault case, the site usually will not be able to recover by itself and a global reconfiguration decision is necessary. All *version faults* will manifest themselves as either "incorrect results", or "missing results".

For example, a missing result from a site can be caused by an erroneous version, which is in an infinite loop, a deadlocked operating system, a hardware fault causing an error in the DEDIX software, etc. A missing result at a site might also be caused by an excessive communication delay, i.e., the result is produced but does not reach the other sites in time. In this case, the sending site will detect the discrepancy between what it sent and what the other sites observed.

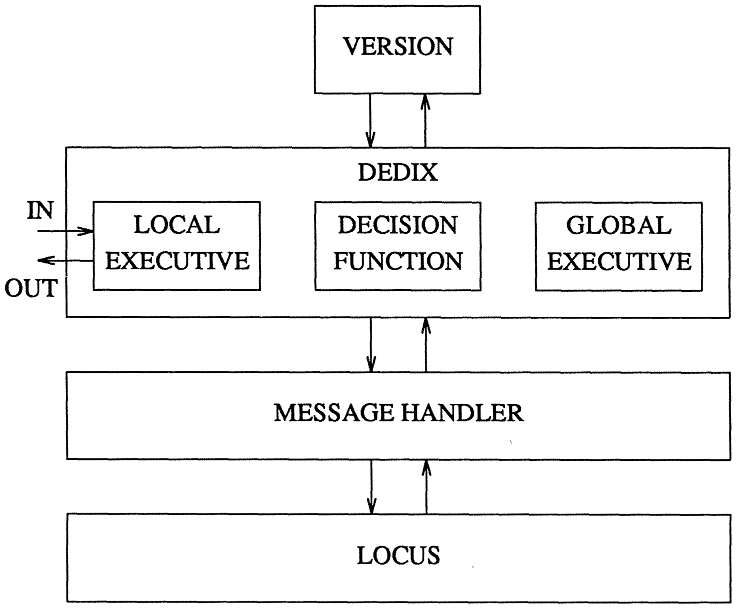

Fig. 2. Functional Structure of DEDIX

Structure. The services of DEDIX are partitioned into three functional modules as shown in Fig. 2 and described in more detail next. They are:

- a *Local Executive* (LE), which is the DEDIX interface to the local version and provides input/output facilities. It also supervises local recovery or reconfiguration actions for the local version.
- a *Decision Function* (DF), which compares a set of results from the different versions to produce a decision result, which is either a consensus result or a decision that a consensus does not exist.
- a *Global Executive* (GE), which monitors the execution at all sites and versions and supervises global recovery and reconfiguration in case

of version failures or a lack of consensus.

2.2 The Local Executive

The Local Executive (LE) contains the DEDIX interface to the version. The version interacts with DEDIX via calls to cross-check functions (cc-functions) and recovery points [Tso 1987b]. The incorporation of these calls is the main adjustment a user has to make in comparison to running his program in a normal (single version) environment. The exact form of these calls is described in Section 3.2. The point of interaction is called cross-check point (cc-point) and the transfered information accordingly cross-check vector (cc-vector).

At cc-points, the cc-functions take the results from the version in form of a cc-vector, translate them to a standard format and pass them to the Decision Function after adding some more identification information. The consensus results produced by the Decision Function are passed back to the disagreeing versions by the LE for recovery. Input and output are also handled by the Local Executive. Furthermore, the LE has some fault tolerance features.

When the Decision Function indicates that the consensus result is not unanimous, or when some unrecoverable exception is signaled from the local version or some other source, the LE will try to recover locally from the fault, report the problem to the Global Executive and, if it is considered as fatal to the site, shut down the site. There are three classes of exceptions that are considered, as discussed below.

Functional exceptions are specified in the functional description of DEDIX and are independent of the implementation. Among them are deviations from an unanimous result, the case when a communication link is disconnected, and the case when a cc-vector is completely missing. For these exceptions the Local Executive will attempt to keep the site active, possibly terminating the local version, while keeping the input/output operating.

Implementation exceptions are dependent on the specific computer system, language, and implementation technique chosen. All UNIX signals, such as segmentation faults, process termination, invalid system call, etc., belong to this class. Other examples are all the exceptions defined in DEDIX, such as signaling when a function is called with an invalid parameter, or when an inconsistent state exists. Most of these exceptions will force an orderly shutdown of a site in order to be able to provide data for analysis.

Exceptions generated by the local version. The local version program is likely to include facilities for exception handling, and some of the exceptions may not be recoverable within the version. These exceptions are sent to the Local Executive which will terminate the local version, while keeping the site alive.

2.3 The Decision Function

The Decision Function is used to determine a single consensus result from the N version results. The Decision Function may utilize only a subset of all N results for a decision; for example, the first result that passes an acceptance test may be chosen. In case a consensus result cannot be determined, a higher level recovery procedure needs to be invoked, that is determined by the Global Executive.

DEDIX provides a generic decision algorithm which may be replaced by the user's custom algorithm, provided that the interfaces are preserved. This allows application-specific decision algorithms to be incorporated in those cases where the standard decisions are inappropriate, or insufficiently precise.

The current decision algorithm searches for a consensus by applying one of the following comparisons to the version results:

(1) **exact (bit by bit)** - allowing an identical match only;

(2) **numerical** - integer and real number comparisons with an allowed range of deviation for "similar" results;

(3) with **"cosmetic"** corrections - allowing for minor (defined) character string differences that may be caused by misspelling or character substitution.

2.4 The Global Executive

The Global Executive (GE) is activated when a recovery point (r-point) is executed. Each r-point has a unique r-point identifier (rp-id) [Tso 1987b]. At first, the GE performs the following actions to determine if global recovery is necessary: 1) compares the rp-ids delivered by the versions, 2) exchanges error reports with other Global Executives, and 3) determines which versions have failed, i.e., disagree with the consensus.

Error reports. Every Global Executive has an error report table, with one entry for each site. This entry is an error counter for that site. The GE increments the counter for a site, whenever that site has either a disagreeing or missing result at a cc-point. This means that the GE does distinguish between a missing result and a delayed result. Since sites might get different numbers of results due to varying communication delays, sites may have somewhat different error reports. The exchange and comparison of error reports ensure a consensus among the GEs at various sites on which versions have failed. If no failed version is detected, the GE merely resets the error report table and the versions continue their execution. Otherwise, global error recovery is initiated.

Two types of failed versions are distinguished: 1) those that have detected errors at the cc-points, and 2) those with incorrect or missing rp-ids. Each Global Executive of a good version signals the *state-output exception handler* of its local version to output the internal state at that rp-id. These states are compared by the Decision Function to obtain the Decision State. Each failed version of the first type is recovered by invoking its *state-input exception handler* to input the Decision State. After the exchange of internal states, actions of the global error recovery are completed and execution of the versions is resumed. A failed version of the second type is first aborted by its Global Executive. The version is then restarted by its GE at the r-point with the decision rp-id. The restarted version also inputs the Decision State by invoking the state-input exception handler before its execution is resumed.

The reconfiguration decision. If a version has produced errors at two or more consecutive r-points, reconfiguration (by shutdown) needs to be initiated. If the shutdown applies to a site, each Local Executive instructs its message receivers to stop receiving from that site. If the shutdown applies to a version, its Local Executive terminates the local version and stops sending messages. In both cases, the new number of expected results is adjusted accordingly by the Decision Function at all sites. After a version is shut down, the site will still collect messages and operate input/output, but it will not deliver them to the local version. The Decision Function and the Global Executive at a site are not affected if only the local version is shut down.

2.5 The Message Handler

Since Locus does not supply all the message handling routines needed for DEDIX, an interface between the described three functional modules of DEDIX and the Locus operating system is necessary. This message handler (MH) interface consists of two layers: the *Synchronization layer* and the message *Transport layer*, where the Synchronization layer is the DEDIX-dependent part of the message handler, while the message Transport layer is DEDIX-independent and depends on the service provided by the underlying operating system.

2.5.1 The Synchronization Layer

For each physically distributed site, the Synchronization Layer (SL) broadcasts results (using the Transport Layer) and collects messages with the version results ("cc-vectors") from all other sites. The SL only accepts results that are both broadcast within a certain time interval and that arrive within the same time interval. The collected results are delivered to the functional modules at the site. The SL accepts a new set of version results when every site has confirmed that all or enough of the previous results have been delivered.

The sites of DEDIX need to be event-synchronized in order to ensure that results from corresponding cc-points are compared. Otherwise, if two sets of results from two different cc-points are compared, the Decision Function might wrongly conclude that some of the versions or sites are faulty. Traditionally, this synchronization has been obtained by referring to a common clock or set of clocks. The SIFT system [Melliar-Smith 1982] is one example of such a clock synchronous system. In SIFT it is predicted *when* the results should be available for a comparison. To ensure that the results are available in SIFT, several design measures are taken to eliminate all unpredictable delays, such as using a fully connected communication structure, using strict periodic scheduling, not allowing external interrupts (only clock interrupts are allowed for scheduling), and regularly synchronizing the clocks.

The local network system and the diverse versions have the following characteristics which make the clock synchronous technique impractical in DEDIX:

- the versions can have different *execution times* between the cross-check points;
- the versions *will run concurrently* with other network activities, which means that sites temporarily can be heavily loaded, and hence prolong the time to execute some versions;
- the Ethernet communication network has inherently varying message transport delays.

A synchronization protocol is designed to provide the event-synchronization service. It ensures that the results that are compared by the Decision Function are from the same cross-check point (cc-point) in each version. The versions are halted until all of them have reached the same cc-point, and they are not started again until the results are exchanged and a decision is made. To be able to detect versions that are in an infinite loop, or otherwise too slow, a time-out technique is used by the protocol.

The use of this synchronization protocol is based on the assumptions that:

(a) correctly working versions produce exactly the same number of cc-vectors in the same order;

(b) correctly working versions have compatible execution times, i.e., they will produce results within a specified time-out interval;

(c) "missing" or disagreeing results do not exist for a majority of versions.

The specification and verification of the protocol is described in [Gunningberg 1985].

Time-out function. The only way to detect that a version did not produce a result when it was supposed to, or that the result is "stuck" somewhere in the communication system, is to use a time-out function, i.e., to require that every version must produce a result within a time-out interval. Two time-out techniques have been considered. The first technique is similar to the time-acceptance test in the recovery block technique. A time-out function is started at the beginning of each segment of computation and all versions must produce results within the specified time interval in order to pass the time-acceptance test. The length of the interval can either be adjusted to each segment of computation or to a "worst case" interval for all segments.

In the second technique, the time-out interval is started when a majority of results have arrived at a site. For example, the time-out is started when the third result arrives in a configuration with five active versions. This technique is based on a comparison between relative execution times instead of using an absolute time, as in the first technique. The time-out is of course terminated if all results arrive before the time-out interval expires. A malfunctioning version sending results too early will not cause any problems, since they will not start the time-out. Interestingly, the problem is similar to "comparing results with skew": the median number (result number 3 out of 5) constitutes the closest to the "ideal value" and the skew corresponds to the time interval. One advantage with this technique, compared to the previous, is that there is no need to assign an individual time-out for each segment of computation. This is an advantage, since the execution time might depend on an a priori unpredictable input, which might put the computation into a loop of long duration. Furthermore, since the time between different cc-points may vary and the sequence of the cc-points is not predetermined, the synchronization would need complex information to adjust the individual time-out intervals.

Both techniques can exist together in DEDIX, and the choice may depend on the application, the input/output, the computing environment, and the real time requirements. Both techniques require that version computations should start almost at the same time at each site and that user input also must arrive within the defined time interval. In the current DEDIX system, the second technique is implemented, due to the operating environment and the type of computations. The time interval is set by the user and can be quite wide, since all versions are suspended until the time interval has expired or until all results have arrived. This suspension is possible since currently there is no real time requirement within DEDIX. The system would need some modifications to accommodate the time-acceptance test technique.

2.5.2 The Transport Layer

The Transport Layer (TL) controls the communication of messages. It hides the system primitives that are actually employed from its user modules. The TL makes sure that no message is lost, duplicated, damaged, or misaddressed, and it preserves the ordering of sent messages.

The requirements of the Transport Layer are specified in terms of response time, throughput, and reliability of service. In order to satisfy the reliability requirement, in most practical situations a redundant communication structure needs to be used. Currently, a single ring structure of inter-process UNIX pipes is employed due to the limitation on the number of pipes per process. Since this implementation does not tolerate a site crash, a redundant interconnection structure is under implementation at the present time.

2.6 Possible Configurations of DEDIX

Given the fundamental functional modules of DEDIX that were described above (LE: Local Executive; DF: Decision Function; GE: Global Executive; SL: Synchronization Layer; TL: message Transport Layer), several configurations of DEDIX can be implemented.

a) For standard operation all three functions, LE, DF, and GE, reside on the same site, and all are present in the same number as the versions. This leads to the standard DEDIX configuration as shown in Fig. 3, with one SL and one TL servicing all three functions.

b) It is possible to have fewer GE modules than versions, even only one GE, that could reside on a separate site with its own

Synchronization and Transport modules, while the LE and DF are sharing their SL and TL modules, as shown in Fig. 4.

c) It is possible to have DF not associated with each version, e.g., have fewer DFs, or even only one DF. This DF can also be located on a separate site, as shown in Fig. 5.

d) Generally, it is possible to construct specialized hardware units which contain separately the functions LE, DF, and GE. Only the LE module needs to have the capability of running a version.

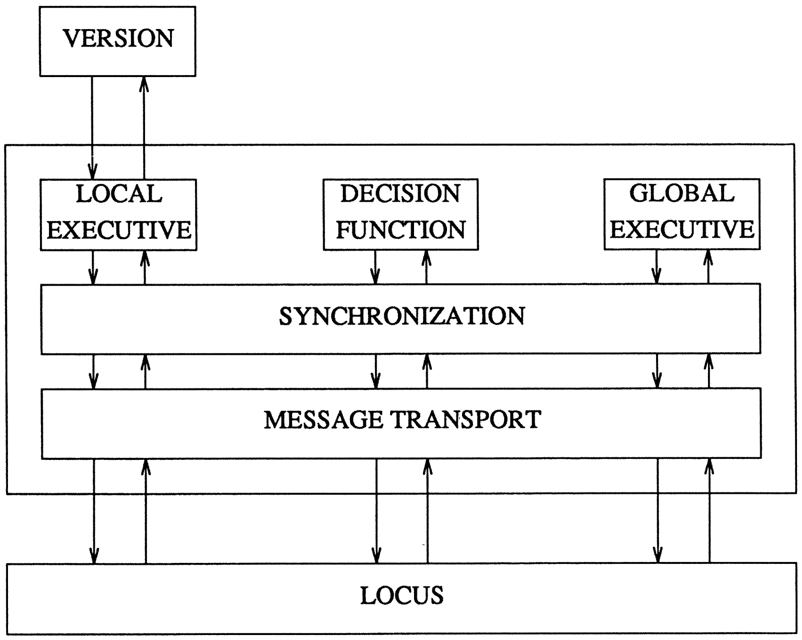

Fig. 3. The Standard DEDIX Configuration on One Site

The structure of DEDIX is adaptable for different applications. Depending on the reliability requirements and the reliability of the versions and the hardware, the decision algorithm and the reconfiguration possibilities, a special arrangement and solution can be chosen. The configurations of Fig. 4 and Fig. 5 have actually been used at UCLA for specific experiments.

3. Current Implementation of DEDIX

A prototype of DEDIX began operation in early 1985 [Avižienis 1985a]. It has been implemented using the C programming language and is running in

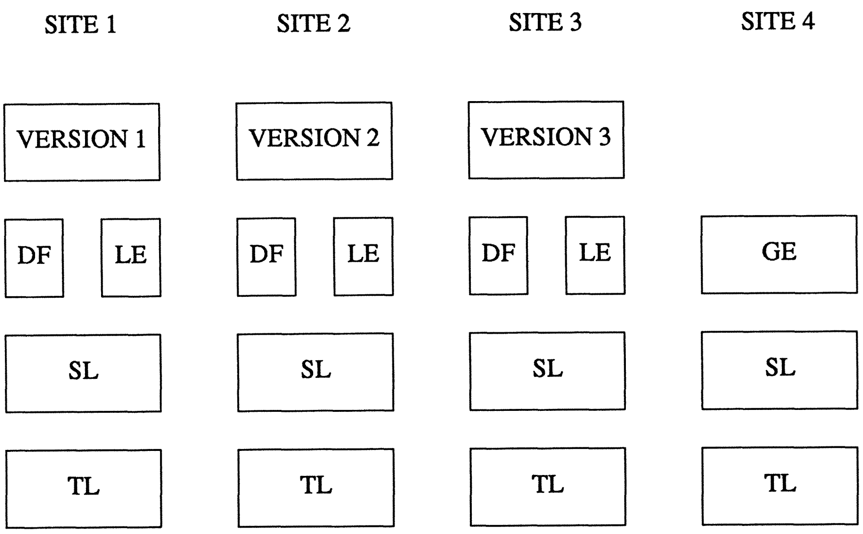

Fig. 4. 3 Sites with 3 Versions, LE, and DF, Site 4 with single GE

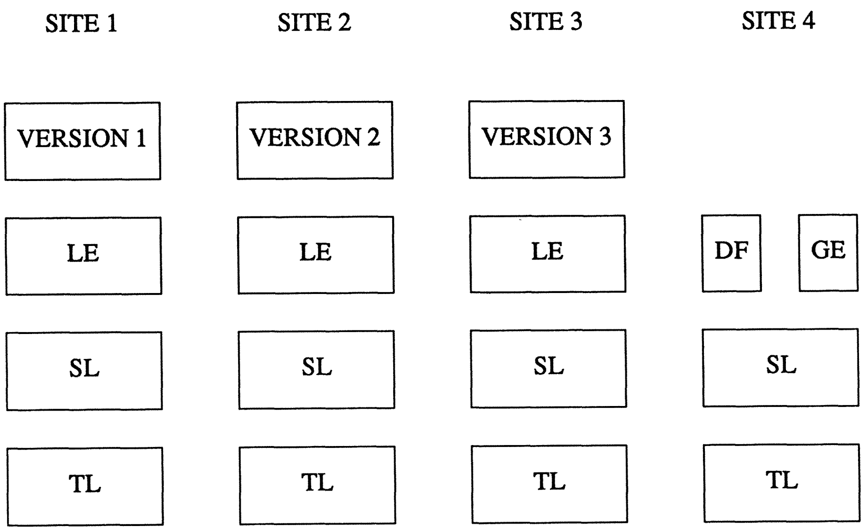

Fig. 5. 3 Sites with 3 Versions, Site 4 with DF and GE

the Locus environment at UCLA. Several modifications and refinements have been incorporated in DEDIX since 1985, mostly to improve the speed of *N*-version execution. The current standard realization has one LE, DF, and GE for each version, running on the same site as the version. It is also possible to execute multiple versions (and therefore also multiple DEDIX replicas) on one site - a single VAX 11/750 machine.

3.1 The User Interface

The user interface of DEDIX allows users to debug the system as well as the versions, to monitor the operations of the system, to apply stimuli to the system, and to collect data during experiments. Several commands are provided to the user, as discussed below.

Breakpoint. The *break* command enables the user to set breakpoints. At a breakpoint, DEDIX stops executing the versions and goes into the user interface, where the user can enter commands to examine the current system states, examine past execution history, or inject faults to the system. The *remove* command deletes breakpoints set by the break command. The *continue* command resumes execution of the versions at a breakpoint. The user may terminate execution using the *quit* command. 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.

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 previously. 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 the decision algorithm, etc.

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

3.2 The Version Interface

Cross-check functions. The programmer of a version must incorporate calls to the Cross-Check Functions (cc-functions) in order to make use of the support provided by DEDIX. These calls have to be included at logically identical cross-check points (cc-points) in the different versions which are going to communicate via DEDIX. Therefore, the cc-points have to be defined in the common specification of the versions. The cc-function calls have the following structure: *ccpoint (ccid, format, arguments).* The parameter list is called *cc-vector.* The *ccid* is the identification of the cc-point. The *format* is a string of characters identifying the types of the variables contained in the arguments and the kind of decision algorithm which is to be applied to these variables. Possible variable types are character, integer, real, etc. The possible decision algorithms are described in Section 2.3.

The identity number of the cross-check point is passed on to the cc-function, to make sure that only information belonging to the same cc-point is compared by the Decision Function. Three different cc-function calls are possible: *ccinput* for input of data (instead of a standard input read statement), *ccoutput* for output of data (instead of a standard output write statement), and *ccpoint* for error recovery.

The ccinput call. The input to the versions is initiated via this call. The Decision Function checks whether all versions agree on the format of the input. In case of a single input, the input is distributed to all versions. In case of multiple versions of input, e.g. by redundant sensors, it is possible that either each version receives its related input, or that the Decision Function checks the inputs and chooses a consensus value for common distribution.

The ccoutput call. All outputs of the N-version software unit must be made via DEDIX and therefore must pass through the Decision Function. The results or data from the versions can be a collection of integers, character strings, and real numbers. Along with this data, a selection can be made on which kind of consensus decision shall be used for the outputs. Different consensus for different parts of the output may be specified.

The ccpoint call. This call is made if a cross-check between the versions is desired, but no input or output is required. Again, a decision is made on the version results, and the consensus result is passed back to the versions.

Recovery points. Complete error recovery of failed versions is performed at recovery points (r-points). Associated with each r-point in each version are: a recovery point id (*rp-id*), which uniquely identifies the r-point, and two exception handlers, the *state-input exception handler* and the *state-output exception handler*, that are required to input and to output the internal state of the version (*version state*) in a specified format. The r-point call has the following structure: *rpoint (rpid)*, where the rpid is the identification of the r-point.

4. Research Applications of DEDIX

The *N*-version software research at UCLA has two major long-term objectives:

(1) to develop the principles of implementation and experimental evaluation of fault-tolerant *N*-version software units; and

(2) to devise and evaluate supervisory systems for the execution of *N*-version software in various environments.

Both objectives are strongly supported by the experimental use and the continuing evolution of the DEDIX supervisory system. Some key aspects of the research applications of DEDIX are discussed below, and some recent specific results - in subsequent sections.

4.1 Implementation and Evaluation of *N*-Version Software

The *N*-version implementation studies that are supported by DEDIX address the problems of: 1) methods of specification, and the verification of specifications; 2) the assurance of independence of versions; 3) partitioning and matching, i.e., good choices of cc-points, r-points, and cc-vectors for a given problem; 4) the means to recover a failed version; 5) efficient methods of modification for *N*-version units; 6) evaluation of effectiveness and of cost; 7) the design of experiments.

Initial specification and partitioning. The most critical condition for the independence of design faults is the existence of a complete and accurate specification of the requirements that are to be met by the diverse designs. This is the "hard core" of this fault tolerance approach. Latent defects, such as inconsistencies, ambiguities, and omissions, in the specification are likely to bias otherwise entirely independent programming or logic design efforts toward related design faults. The most promising approach to the production of the initial specification is the use of formal, very-high-level specification languages that are discussed in Section 6. When such specifications are executable, they can be automatically tested for latent defects and serve as prototypes of the programs suitable for assessing the overall design. With this approach, perfection is required only at the highest level of specification; the rest of the design and implementation process as well as its tools are not required to be perfect, but only as good as possible within existing resource constraints and time limits.

The independent writing and subsequent comparisons of two specifications, using two formal languages, is the next step that is expected to increase the dependability of specifications beyond the present limits. Our current investigation of specification methods is discussed in Section 6. It is also important to note that a formal specification must specify the following application-specific features needed by *N*-version execution: 1) the initial state of the program; 2) the inputs to be received; 3) the location of cross-check and recovery points (partitioning into modules); 4) the content and format of the cross-check vector at each cc-point and r-point (outputs are included here); 5) the algorithms for internal checking and exception handling within each version; and 6) the time constraints to be observed by each program module.

Independence of version design efforts. The approach that is employed to attain independence of design faults in a set of *N* programs is maximal independence of design and implementation efforts. It calls for the use of diverse algorithms, programming languages, compilers, design tools, implementation techniques, test methods, etc. The second condition for independence is the employment of independent (noninteracting) programmers or designers, with diversity in their training and experience. Wide geographical dispersion and diverse ethnic backgrounds may also be desirable. DEDIX provides a suitable environment to study the effectiveness of efforts to attain diversity and independence of versions. Recent experimental results and plans for future experiments are reviewed in Section 5.

Recovery of failed versions. A problem area that has been addressed recently [Tso 1987a] is the recovery of a failed version in order to allow its continued participation in *N*-version execution. Since all versions are likely to contain design faults, it is critically important to recover versions as they fail rather than merely degrade to *N*-1 versions, then *N*-2 versions, and so on to shutdown. Recovery of a given version is difficult because the other (good) versions are not likely to have identical internal states; they may differ drastically in internal structure while satisfying the specification. The Community Error Recovery (CER) approach offers a systematic two-level method of forward recovery for failed versions [Tso 1987b]. Recent results of an experimental evaluation of CER using DEDIX are presented in Section 5.

Modification of *N*-version software. It is evident that the modification of software that exists in multiple versions is more difficult. The specification is expected to be sufficiently modular so that a given modification will affect only a few modules. The extent to which each module is affected can then be used to determine whether the existing versions should be modified according to a specification of change, or the existing versions should be discarded and new versions generated from the appropriately modified specification. DEDIX-based experiments are currently being planned to gain insights into the criteria to be used for a choice.

Assessment of effectiveness. The usefulness of the *N*-version approach depends on the validity of the conjecture that residual software faults in separate versions will cause very few, if any, similar errors at the same cc-points. Large-scale experiments need to be carried out in order to gain evidence on the nature of faults encountered in independently developed program versions. The "mail order software" approach offers significant promise to provide versions to be evaluated using DEDIX. An "international mail order" experiment is being planned, in which the members of research groups from several countries will use a formal specification to write software versions. It is expected that the software versions produced at widely separated locations, by programmers with different training and experience who use different programming languages, will contain substantial design diversity. In further experiments, it may be possible to utilize the rapidly growing population of free-lance programmers on a contractual basis to provide module versions at their own locations. This approach would avoid the need to concentrate programming specialists, have a low overhead cost, and readily allow for the withdrawal of individual programmers.

Cost investigations. The generation of *N* versions of a given program instead of a single one shows an immediate increase in the cost of software prior to the verification and validation phase. The question is whether the subsequent cost will be reduced because of the ability to employ two (or more) versions to attain mutual validation under operational conditions. Cost advantages may accrue because of 1) the faster operational deployment of new software; and 2) replacement of costly verification and validation tools and operations by a generic *N*-version environment (such as DEDIX) in which the versions validate each other while executing useful work. The loss of performance due to the presence of fault tolerance mechanisms, such as decision algorithms and recovery points also needs to be assessed.

Design of experiments. Several design issues of design diversity experiments need to be carefully resolved. They include: exploring different dimensions of diversity, incorporating efficient error detection and recovery algorithms, and avoiding commonalities in the design effort. The software versions produced in these experiments need to be subject to controlled conditions that approximate the development methodologies and environments used by advanced industrial facilities. There should be extensive logging of work periods and events such as error detection, specification questions and answers, and test suite execution. The experiment leaders need to provide a complete high-level, high-quality specification. At all stages, questions about the specifications are submitted by electronic mail, reviewed by the experiment leaders, and answered by electronic mail. The rule of "written communication only" makes it possible to control and analyze the information flow. The determination that a question revealed a flaw in the specifications causes changes to be broadcast to all programmers at all sites. The deliverable items include a design document, a series of compiled programs representing the results of the top down development at each abstraction layer, a test plan and test log, and the final program. The delivered software is then subjected to an adequate acceptance test to ensure its quality.

To measure the extent of design diversity and to assess potential reliability increases under large-scale, controlled experimental conditions, two major projects are underway: The NASA/Four-University experiment (initiated in the summer of 1985) and the UCLA/Honeywell Experiment (to be conducted during the second half of 1987). Descriptions of these two experiments are presented in the following Section 5.

4.2 Investigations of Supervisory Systems

The research concerned with *N*-version supervisory systems, as exemplified by DEDIX, deals with: 1) the functional structure of supervisors; 2) fault-tolerant supervisor implementation, including tolerance of design faults; 3) instrumentation to support *N*-version software experiments; 4) efficient implementation, including custom hardware architectures to support real-time execution; and 5) methods of supervisor evaluation.

***N*-version execution supervision and support.** Implementation of *N*-version fault-tolerant software requires special support mechanisms that need to be specified, implemented, and protected against failures due to physical or design faults. These mechanisms fall into two categories: those *specific* for the application program being implemented, and those that are *generic* for the *N*-version approach. The *specific* support is part of the version specification. The *generic* class of support mechanisms forms the *N*-version execution supervision environment that includes: 1) the decision algorithm; 2) assurance of input consistency; 3) interversion communication; 4) version synchronization and enforcement of timing constraints; 5) local supervision for each version; 6) the global executive and decision function for the recovery or shutdown of faulty versions; and 7) the user interface for observation, debugging, injection of stimuli, and data collection during *N*-

version execution of application programs. The nature of the generic support mechanisms has been illustrated in the discussions of the DEDIX N-version supervisor system that was described in preceding Sections 2 and 3. The continuing use of DEDIX leads to further insights that result in refinements and enhancements of DEDIX functional structure and its efficiency.

Protection of the supervisory environment. The success of design fault tolerance by means of N-version software depends on uninterrupted and fault-free service by the N-version supervision and support environment. Protection against physical faults is provided by the physical distribution of N versions on separate machines and by the implementation of fault-tolerant communication linkages. The SIFT system [Wensley 1978] and DEDIX are suitable examples in which the global executive is also protected by N-fold replication. The remaining problem is the protection against design faults that may exist in the support environment itself. This may be accomplished by N-fold diverse implementation of the supervisor. To explore the feasibility of this approach, the prototype DEDIX is currently undergoing formal specification. Subsequently, this specification will be used to generate diverse multiple versions of the DEDIX software to reside on separate physical nodes of the system. The practicality and efficiency of the approach remain to be determined. Some results are discussed in Section 6 of this paper.

Architectural support. Current system architectures were not conceived with the goal of N-version execution; therefore, they lack supporting instructions and other features that would make N-version software execution efficient. For example, the special instructions ''take majority vote'' and ''check input consistency'' would be very useful. The practical applicability on N-version software in safety-critical real-time applications hinges on the evolution of custom-tailored instruction sets and supporting architectures. The current DEDIX implementation supported by Locus is likely to be too slow for this purpose. Despite this limitation, the functional architecture of DEDIX can be used with faster transport service and faster scheduling policies in a real-time system, while Locus can be used to simulate real-time execution.

5. Testing Tools, Experience, and Results

5.1 Programs for Demonstration and Testing

This section describes the existing multiversion programs for DEDIX which were developed to test DEDIX and to demonstrate its capabilities. The most important characteristics of these application programs are also given. Most of the programs were written in the C programming language.

The *Airport Scheduler* simulates an airport database and is based on the specification used in [Kelly 1983]. Typical operations include: scheduling or canceling a flight, changing certain flight data (e.g. departure time), reserving a seat, and looking up information in the database. There exist three versions: one version implements the database using arrays, the second one uses linked lists. Both versions are written in C, and the third version is identical to the first, except that it is written in Pascal. These three programs demonstrate the concept of design diversity and are suitable to test DEDIX after modifications.

The programs *arcade* and *arc_io* can be used to test the implementation of the cross-check functions. A number of borderline cases are explored. Furthermore, calls which violate the specification of these functions are made in order to test the robustness of DEDIX. Examples are: calls without cross-check vectors, with inconsistent cc-point identifier, wrong format string, or inconsistencies between format string and cc-vector.

The program *cv* tests the implementation of the Decision Function. It reads test cases from a file, applies them to the decision algorithm, and stores the decision result in an output file. A file with some 100 standard test cases is available, as are the expected results from these test cases. Three versions exist which differ in that they read different files.

Name is a sample demonstration program that reads an integer, performs some "complicated" computations involving arrays of integer and real numbers, and that finally selects a string (name) from a table for display. Four versions exist: three mutated ones with different built-in data tables, and one that simulates an infinite loop.

The program *power* computes a to the power of b for two built-in numbers a and b. It is designed to exercise the number-handling features of DEDIX. Twenty different bugs (e.g. different numerical constants, typing errors, wrong use of cc-functions) have been injected into the program and can be invoked. Thus 2^{20} different versions can be generated. These versions serve as mutants for the validation of DEDIX by mutation testing.

Test is a test program to test the basic functions of DEDIX. It is similar to *arcade* and *arc_io*, but is better documented and tests more thoroughly. Standard test cases and their expected results are available. In addition, there exists a shell script that repeatedly executes these test cases in different configurations (single machine, distributed) or with different run time options of DEDIX.

Time is a demonstration program that reads the clock of the machine it is running on, converts it into a string and displays it. It also asks interactively whether that process should be repeated. *Time* demonstrates that diverse versions can have synchronized clocks. Naturally, this program is only useful when DEDIX is distributed, i.e. each version runs on a different machine.

Table 1 summarizes the characteristics of the above mentioned testing programs.

Table 1. Multiversion Programs for Demonstration and Testing Purposes

Name	Number of Versions	Lines of Code (approx.)	Language	Main Purpose
airport	3	470	C, Pascal	demonstration
arcade, arc_io	6	350	C	test - cc-functions
cv	3	100	C	test - voter
name	4	60	C	demonstration
power	20+	515	C	test - number handling, mutation testing
test	4	300	C	test - all basic functions
time	7	20	C	demonstration

5.2 Proper Specification and Testing of Fault Tolerance Mechanisms

It was noted earlier that the *N*-version error detection and recovery mechanisms for each version, including cross-check points and recovery points, need to be defined in the software requirement specification. To avoid restricting design diversity, the programmers may be given a choice where to place the cc-points in their programs. The sequence in which the cc-points occur and the variables involved should be specified, and it should be required that the variables of each cc-point be computed but not used before the cc-point is reached. The programmers are also required to use the (possibly modified) values returned by the DEDIX supervisor in all subsequent computations.

The acceptance test should adequately test the recovery capability. It should ensure that the cc-points are placed in the right sequence, and output values are checked in right places during the execution of each version. Possible design faults that are related to cc-points fall into two categories:

1. *Incorrectly located cc-points.* Some programmers might place cc-points before the final values are calculated. These cc-points are placed too early. Also, some versions might use computed values before passing them to the decision function. These cc-points will occur too late.

2. *Unused returned values.* This fault could occur when a version uses an internal variable in place of a state variable. The value of the internal variable is assigned to the state variable of the cc-vector before the cc-point is called, but subsequent computations are still based on the value of that internal variable.

These faults can be detected by specially designed tests. The output values should be checked at the cc-points. This will detect the incorrect placement of cc-points. Also, specific tests should be included that deliberately return new values to some cc-points. The results of the next cc-point should then be checked to verify that the returned values are actually used. These preparations are necessary for the proper execution of multi-version software in the DEDIX environment.

5.3 DEDIX in the NASA/Four-University Multiversion Software Experiment

The NASA Langley Research Center is sponsoring the NASA/Four-University experiment in fault-tolerant software which has been underway since 1984. During the summer of 1985, the NASA experiment employed 40 graduate students at four universities to design, code and document 20 diverse software versions of a program to manage redundancy and to compute accelerations for a redundant strapped down inertial measurement unit (RSDIMU). The analysis of this software currently engages researchers at six sites: UCLA, the University of Illinois at Urbana-Champaign, North Carolina State University, and the University of Virginia, as well as the Research Triangle Institute (RTI), and Charles River Analytics (CRA). Empirical results from this experiment will be jointly published by the cooperating institutions after the verification, certification, and final analysis phases are complete. While the joint results still await publication, some independent results from the UCLA effort have been reported in [Kelly

1986].

During the summer of 1985, each of the four universities employed ten graduate students to design, code and document five software versions in ten weeks. At the end of this effort, each of these 20 software versions was required to pass a preliminary acceptance test that used 75 test cases. At UCLA, a long and careful validation phase including extensive testing of the versions followed the 10-week software generation phase. During validation, many errors and ambiguities in the specifications and the software versions were revealed. The specifications were subsequently refined. The five UCLA versions have since been further debugged by the original programmers and have passed a final (UCLA) certification test that consisted of 200 random test cases, 55 hand-made test cases of special value test data and extremal value test data, and special test cases for verifying the recovery mechanism. The size of the five resulting software versions ranged from 1677 to 2794 lines of Pascal statements. The scope of this discussion is limited to the specific testing done at UCLA that employed DEDIX. The purpose of the tests was to evaluate the new CER forward recovery method [Tso 1987b]. Only the five certified versions from UCLA were used in these tests.

A Test Case Generator (TCG) was used throughout the evaluation of recovery to generate random test cases. After the TCG had generated the data for a test case, all five individual versions were executed consecutively, using the same input data. If a majority of similar results exists, they are used to decide the *reference* output which is further checked by the known TCG output values to ensure its consistency. At the same time, individual version failures are identified. This failure information is used to generate "interesting" 3-version combinations (triplets) using the assumption that all majority versions behave identically for that test case. This means that triplets with two good versions, such as (G1, G2, B), (G1, G3, B), and etc., are treated as one, i.e., (G, G, B), and many triplets can be eliminated from further testing. The *interesting* triplets are then executed in a three-version configuration under DEDIX supervision. The decision results are passed back to the failed versions for partial recovery at the cc-point level. Decision results of the triplets without recovery are obtained simply by comparing individual version outputs of the combinations. The decision results, both a) *without recovery* and b) *with recovery*, are then used to determine the effectiveness of the recovery. The process then is repeated for further test cases. A total of 200,000 test cases were employed in recovery evaluation.

5.3.1 Faults Discovered and Errors Observed During Testing

During the recovery evaluation process, several faults were found in the five UCLA certified versions. Table 2 lists these faults and their effects on the outputs, i.e., the errors seen at cc-points.

The fault ucla1-1 manifested itself during the testing because of the use of a Pascal compiler in the testing harness, while a Pascal interpreter was used in the program development and certification processes. Obviously, the interpreter initializes variables in a Pascal procedure, while the compiler does not. Since this fault failed the version more than half of the time, it was taken out in our evaluation. One of the display functions is to display the five most significant digits and the decimal point of a floating point number. Two versions failed to round the numbers correctly, although not in the same way. Both versions ucla3 and ucla4 made wrong system failure decisions, but for two different reasons. Thus the faults are different, but both versions produced coincident and identical errors at the cc-point for 96 out of the 200,000 test cases.

Failures of the individual versions. The result of a version running a test case is defined as erroneous if one or more of its output values (out of a total of 64 element values) differs from the *reference* values defined previously. We also say that the version *fails* on that test case. Table 3 shows the observed failures for the *individual* program versions for the 200,000 test cases, and their sizes in number of Pascal statements.

It must be noted that the failure probability depended very much on the test case generator, and on the range of variation ("skew") that is allowed when results are compared. We consider that the versions tested in this evaluation were under stress because the test cases were sampled randomly from the largest possible input space. In actual flight, extremal input data are much less likely to happen than routine data.

Coincident failures of the versions. Two versions are said to fail coincidently if they both fail (produce erroneous values of the same element) for the same test case. These coincident errors may be similar or distinct. It was observed that more than two versions did not fail for the same test case during the 200,000 test runs. There was one coincident error between ucla1 and ucla3, and there were 110 coincident errors between ucla3 and ucla4.

Similar errors of the versions. It should be noted that the results of the versions which fail coincidently may not be similar. *Similar results* are defined to be two or more results (good or erroneous) that are within the

Table 2. Characteristics of Discovered Faults

Label	Class	Fault	Error at cc-point
ucla1-1	incorrect algorithm	uninitialized variable	incorrect sensor status
ucla1-2	incorrect algorithm	bad display rounding	incorrect display
ucla1-3	incorrect algorithm	overflow handled incorrectly	incorrect display
ucla2	no fault discovered		
ucla3-1	spec mis-interpretation	individual instead of average slopes used	incorrect sensor status
ucla3-2	spec mis-interpretation	wrong frame of reference used	incorrect sensor status
ucla3-3	spec ambiguity	wrong system failure decision	incorrect system status
ucla3-4	incorrect algorithm	bad display rounding	incorrect display
ucla3-5	incorrect algorithm	overflow not handled	incorrect display
ucla4-1	spec ambiguity	wrong system failure decision	incorrect system status
ucla5	no fault discovered		

range of variation that is allowed by the decision algorithm. When two or more similar results are erroneous, they are called *similar errors* [Avižienis 1985b]. It was found that only ucla3 and ucla4 had similar errors which occurred for 96 test cases.

5.3.2 Results of CC-Point and R-Point Recovery

The effectiveness of partial recovery at cc-points was evaluated by

Table 3. Failures of Individual Versions

Version	Size	Number of Failures	Failure Probability
ucla1	2016	1	0.000005
ucla2	1685	0	0.000000
ucla3	1962	702	0.003510
ucla4	2794	283	0.001415
ucla5	1677	0	0.000000

comparing the *final decision* results of a 3-version RSDIMU software module (triplet) executed *without* the cc-point recovery provision and the one executed *with* the cc-point recovery provision. This was the first opportunity to perform cc-point recovery as an experiment.

The *final decision* of a triplet falls into five categories as shown in Table 4.

Table 4. Classification of Triplet Decisions

Final Decision	Individual Version Results			Explanation
GOOD3	G	G	G	All three results are good (G).
GOOD2	G	G	B	Only two results are good. The error (B) of the failed version is masked.
NOMAJ	B1 B1	B2 B2	G B3	All three results are different from each other. This decision is a fail-safe stop.
BAD2	B B	B B	G B1	A similar error (B) occurs in two versions.
BAD3	B	B	B	A similar error in all three versions.

Table 5 summarizes the consequences of including the cc-point recovery over the 200,000 test cases. Almost 90% of the changed decisions of the 3-version RSDIMU module are from GOOD2 to GOOD3, meaning that errors which occurred in a single version of the triplets had been recovered successfully by cc-point recovery. The improvement of a decision from GOOD2 to GOOD3 should not be diminished by the fact that the decision results of the two decisions are the same, and the change is only on the confidence level. This improvement makes the 3-version MVS system fully recovered and ready to tolerate another fault that may happen in the subsequent computations. There are 64 decisions in the GOOD3 → GOOD3 category although the triplets include one or two failed versions. This occurs because our analysis considers the System Status and Estimated Acceleration results only, and these failed versions were able to compute them correctly, but failed in the Display Driver.

Table 5. Consequences of CC-Point Recovery

Without recovery	With recovery	Triplets of 2 G and 1 B versions	Triplets of 1 G and 2 B versions	Total	Percent
GOOD3 →	GOOD3	63	1	64	–
GOOD2 →	GOOD3	923	3	926	89.6
NOMAJ →	GOOD3	0	9	9	0.9
NOMAJ →	BAD3	0	2	2	0.2
BAD2 →	BAD3	0	96	96	9.3
Total number of changed 3-version RSDIMU module results				1033	100

There are nine triplets which had their decisions improved from NOMAJ to GOOD3. This improvement happens when two different versions fail at different cc-points. Without recovery, the triplet produced a NOMAJ decision; with recovery, it first recovered a failed version at an earlier cc-point, then the fully recovered triplet recovered another failure later. Since the RSDIMU module has only five computations, and most of the observed errors occurred after the second one, the case in which a fully recovered triplet recovered from a second fault happened rather rarely.

In the 96 triplets that had their decisions changed from BAD2 to BAD3 the good version was forced to fail in the same way by an attempted recovery. However, similar errors already existed in a majority of versions, and the MVS system is assumed to fail, in either case. The two triplets that have their decisions changed from NOMAJ to BAD3 are dangerous because the 3-version MVS system has been changed from a fail-safe state to an unsafe state.

The most frequent similar errors observed during the testing are due to the case in which both versions ucla3 and ucla4 declare that the RSDIMU system failed. This decision sets all sensor status to non-operative and the estimated accelerations to zero. The program faults (ucla3-3 and ucla4-1 in

Table 2) are due to extra checks on conditions that should not happen according to the original RSDIMU specification. This specification was changed during the course of program development and certification. However, it should be noted that such outputs lead to a fail-safe response of shutting down the system in the RSDIMU application. Detailed discussion of the results appears in [Tso 1987b].

Recovery points were not specified in the RSDIMU specification. However, a new program can be easily composed in which the RSDIMU module is the first module, with an auxiliary (AUX) module added. Then a recovery point is inserted between them. The AUX module contains nothing but a new cc-point used to check if the AUX module is indeed reached and started with a correct version state. The version state at the beginning of the AUX module was defined to be the collection of all the eleven output variables and of two other variables in the RSDIMU module found to be common to all the versions. One of them is the id number of the failed face, and the other is the threshold that determines a sensor failure. With the version state defined, state input and output exception handlers were implemented and used by all five versions.

DEDIX was used for testing because recovery at the recovery point level requires a sophisticated N-version supervisor to keep track of errors detected at the cc-points, to invoke the exception handlers, and to restart an aborted version. All the test cases that caused some versions to fail during previous cc-point recovery testing were used to test triplets of the instrumented programs for r-point recovery. The evaluation is similar to the previous one that considered RSDIMU system improvement with cc-point recovery. The previous evaluation examined the final results (System Status and Estimated Acceleration) of a 3-version RSDIMU module. In this evaluation we examined the version state after the recovery point.

The results of all possible consequences of a DEDIX test run executing a triplet of instrumented versions (containing either one, or two bad versions) are shown in Table 6. Each version consists of the RSDIMU module, an r-point, and the AUX module. Table 6 shows that for triplets with only one bad version, 983 of the 986 version states of the bad versions were recovered correctly at the recovery point, and there were 3 cases in which DEDIX gave the "No Majority" decision because of disagreement in comparing version states. The good versions used to form the triplet were the first two good versions chosen in the order of their version identifiers, therefore they always were different versions. It was found in those 3 test runs that although ucla1 had produced good outputs at the cc-points in the RSDIMU module, in fact it had an erroneous internal state that was revealed by the two additional variables included in the version state specification.

All 98 triplets of one good and two bad versions that had produced BAD3 decisions with cc-point recovery (see Table 5) had the "No Majority" decision while comparing the version states. This happened because the two versions both had incorrectly concluded for different reasons that the RSDIMU module failed, and thus had produced *similar errors at the cc-point that were due to different faults*. However, the two common non-output variables differed and therefore a BAD3 majority decision was avoided. This occurrence shows that r-point checking is more effective than only cc-point checking since the BAD3 cc-point decisions were properly detected at the r-point.

Table 6: Consequences of R-Point Recovery

Possible Consequence	**Triplets with 1 B Version**	**Triplets with 2 B Versions**	**Total**
No majority at the cc-points in the RSDIMU module	0	0	0
No majority in comparing the r-point ids	0	0	0
No majority in comparing the version states at r-point	3	108	111
GOOD3 decision at the cc-point in the AUX module	983	3	986
GOOD2 decision at the cc-point in the AUX module	0	0	0
NOMAJ decision at the cc-point in the AUX module	0	0	0
BAD2 decision at the cc-point in the AUX module	0	0	0
BAD3 decision at cc-point in the AUX module	0	0	0

There are also three test runs of triplets with 2 bad versions that produced the GOOD3 decision at the last cc-point. This happened because the errors of the two bad versions had occurred at different cc-points and were successfully recovered by cc-point recovery (Table 5).

Since control flow errors were not observed in the 200,000 test runs, more testing was conducted through error seeding. The goal was to verify the effectiveness of the restart mechanism of the r-point. Faults of the following two categories were seeded: 1) *Program exceptions*, such as "division by zero" and "index out of range," and 2) *control flow faults*, such as "infinite loop" and "incorrect branching" that lead to some cc-point being incorrectly called or skipped.

Most of the faults that were seeded into the versions were chosen from faults that were eliminated during the certification process. Testing was conducted with triplets consisting of two good versions combined with a version with a seeded fault. It was found that in all the hundred different test runs that were performed, the failed versions were restarted with a correct version state after the recovery point.

5.4 The UCLA/Honeywell Fault-Tolerant Software Experiment

To gain further insights into the effectiveness and methodology of applying multi-version software systems, UCLA and the Honeywell - Sperry Commercial Flight Systems Division have agreed to conduct a joint study of multi-version software design during the second half of 1987. The application is the digital flight control system for future commercial airliners, as exemplified by the system being developed by Honeywell for potential use in the McDonnel-Douglas MD-11 aircraft.

The objectives of the UCLA/Honeywell project to study the *N*-version flight control system design are as follows:

- To conduct studies and experiments related as closely as practical to the industrial environment in terms of procedures and types of problems.
- To develop a practical and effective set of ground rules for multi-version software development in an industry environment. These ground rules will be directed toward the elimination of significant similar errors in the versions.
- To estimate the effectiveness of multi-version software in an industrial environment of a specified type.

The extent and purpose of the multiversion software is:

(a) The software provides automatic pitch control of commercial aircraft during final approach.

(b) The elements of the control loop are control law, airplane, sensors mounted on airplane, landing geometry, and wind disturbances.

(c) Independent two-programmer teams will program the control laws, based on a software requirements document, i.e., the software specification.

(d) The aircraft and wind turbulence are to be modeled on VAX machines. The operation of flight simulation will be monitored by DEDIX to observe the execution of a multi-version software system.

For the software development phase, six teams of two graduate student programmers each will work in the software development phase for 12 weeks during the summer of 1987. Software engineering techniques to build high quality software will be strictly followed. The six teams will be coordinated by the UCLA research team, using an electronic mail communication facility. A standard industrial design review, code review, and a test review will be conducted. The expected length of code produced in this software development phase should exceed 2000 lines.

Several dimensions of design diversity have been considered for achieving diversity among these programs. The attention will be focused on the use of different programming languages and their effects on the diversity in multi-version software. The languages are C, Pascal, Modula-2, Ada, Lisp, and Prolog.

In the evaluation of resulting multi-version software systems, closed loop testing of multiple executions with random inputs will be conducted. Millions of test runs will be executed in the DEDIX environment for suitable aircraft control and flight simulation. Statistical data related to execution of multi-version software systems will be gathered for the evaluation of the effectiveness of DEDIX.

6. Specification Issues

Significant progress has occurred in the development of formal specification languages, methods, and tools since our previous experiments [Kelly 1983, Avižienis 1984]. Our current goal is to compare and assess the applicability to practical use by application programmers of several formal program specification methods. The leading candidates are:

(1) The Larch family of specification languages developed at MIT and the DEC Western Research Center [Guttag 1985];

(2) The OBJ specification language developed at UCLA and SRI International [Goguen 1979];

(3) The Ina Jo specification language developed at SDC [Locasso 1980];

(4) The executable specification language "PAISLey" developed at AT&T Bell Laboratories [Zave 1986].

The study focuses on the assessment of the following aspects of the specification languages: (1) The purpose and scope, i.e., the problem domain; (2) completeness of development; (3) quality and extent of documentation; (4) existence of support tools and environments; (5) executability and suitability for rapid prototyping; (6) provisions of notation to express timing constraints and concurrency; (7) methods of specification for exception handling; and (8) extensibility to specify the special attributes of fault-tolerant multi-version software.

The goal of the study is the selection of two or more specification languages for the subsequent experimental assessment of their applicability in the design of fault-tolerant multi-version software. Two major elements of the experiment will be:

(1) the concurrent mutual verification of two specifications by symbolic execution and mutual interplay;

(2) an assessment of the practical applicability of the specifications, as they are used by application programmers in an N-version software experiment.

The next step in DEDIX development will be a formal specification of parts of the current DEDIX prototype (implemented in C): the Synchronization Layer, the Decision Function, and the Local and Global Executives. Among them, the Larch specifications of the Decision Function [Tai 1986] and of the Synchronization Layer have been constructed. The specification will provide an executable prototype of the DEDIX supervisory system. This functional specification should allow not only the migration to real-time systems, but also the use of multi-version software techniques for the fault-tolerance mechanisms of DEDIX themselves. The goal is a DEDIX system that supports design diversity in application programs and which is itself diverse in design at each site.

Independent specifications of some DEDIX system modules in two or more formal languages will serve to compare the merits of the methods. Further research is planned in the application of *dual* diverse formal specifications to eliminate similar errors that are traceable to specification faults and to increase the dependability of the specifications.

7. Other Current Research Activities

7.1 Improvement of DEDIX

This section discusses some observed deficiencies of DEDIX and offers some thoughts about improving them. Current activities are also mentioned, where appropriate.

The most visible shortcoming of DEDIX is the execution overhead which results in rather long waiting times for the user. There are two possible ways to improve the situation: one is to create a "custom DEDIX" which is tailored to a specific application. Functions that are not needed can be removed, and the versions can be compiled into DEDIX instead of creating another process for each version to be executed. That reduces greatly the amount of time spent with interprocess communication. The second approach is to look for more efficient implementations of these parts of DEDIX that are used most. Due to the layered design it should be relatively easy to replace a layer with a more efficient one, without affecting the others. Since most time is spent on message passing, an investigation of a more efficient implementation of the transport layer is under way.

Another observation is that DEDIX supports only standard input and output. Thus the ability to manipulate files is limited to redirecting the input and the output. Of course, it is possible for a version to use all the file manipulating operations that are provided by the operating system. However, the checking and correcting facilities of DEDIX would be essentially bypassed in this case. Furthermore, different versions may not read and/or write the same file(s) because that would result in an almost certainly unpredictable interaction between the versions. A solution would be to provide cross-check functions for file I/O, similar to those now provided for standard I/O. However, the following considerations lead to the conclusion that this is not too urgent: diverse software is likely only to be required and applied in systems with ultra-high reliability requirements, e.g. autopilots, flight control systems, air traffic control systems, or nuclear power plant control systems. Systems of these kinds are usually computation intensive, rather than data and I/O intensive. Thus it can be expected that it will be sufficient to support standard I/O for most of these systems.

Furthermore, the versions are limited to sequential programs all of which must execute all the specified cross-check points in the same order. In many cases the sequence of cc-points is given by the data flow of the computation to be performed. However, in case there are several independent submodules which could be executed in any order, a specific sequence of these independent computations has to be specified and all versions have to adhere to it. This, to a certain extent, limits the degree of diversity that could be achieved. Presently, neither a study examining whether this restriction is a severe one or not, nor a method to overcome it, exist. Of course, it is easy if DEDIX only *observes* the computed results, without trying to correct them – we just postpone the analysis until all versions have terminated.

7.2 Extension of DEDIX Capabilities

Byzantine faults [Lamport 1982] are defined as faulty behavior that may prevent agreement about the current (global) system state among the sites of a distributed system. Examples of such behavior include:

- sending more or fewer messages than required to by the protocol,
- sending messages too late or too early,
- sending different (inconsistent) information to different sites, or
- maliciously cooperating with another malicious site.

The Synchronization Layer of DEDIX provides considerable protection against the first two examples of faulty behavior. Since the topology of the current implementation is a ring structure, a site cannot send different information to different sites, but it can alter the information that it is supposed to forward. At the present time, DEDIX does not deal with other types of Byzantine (malicious) faults. Methods to tolerate them are known [Lamport 1982] and could be included in the Transport layer. A study is currently in progress that will provide some experimental data on the time and complexity overhead of these methods.

In order to build an elegant, highly reliable system which is tolerant to both hardware and software design faults, a study is in progress how to build a DEDIX system on top of a XEROX Worm environment [Shoch 1982]. The key idea is that the Worms bring a special philosophy to building distributed, fault-tolerant systems. This philosophy gives each individual unit a high degree of autonomy and a desire to complete its task and to take an active part in the activity of the whole system; and further, takes a network service approach to the resources available in the system.

8. Conclusion

This paper has presented an overview of a major effort to develop a research environment for software design diversity research at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory. The complete DEDIX prototype has been implemented, and it is being used to execute, test, and evaluate multiversion software. Some new research efforts also have been initiated.

Acknowledgment

The research described in this paper has been supported by a grant from the Advanced Computer Science program of the U.S. Federal Aviation Administration, by NASA contract NAG1-512, and by NSF grant MCS 81-21696. Professor Algirdas Avižienis, Director of the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory, has served as Principal Investigator since the inception of the DEDIX project.

The original concept and implementation of DEDIX, as described in [Avižienis 1985a], has benefited from major contributions of several individuals who were visiting researchers at UCLA in the 1983-85 period. A large part of the DEDIX implementation is due to Lorenzo Strigini, who is currently at the IEI-CNR, Pisa, Italy. The communication and synchronization protocols are the contribution of Per Gunningberg, presently at the Swedish Institute of Computer Science, Stockholm, Sweden. The original decision function was designed and implemented by Pascal Traverse, now at Aerospatiale, Toulouse, France. John P. J. Kelly, now at the University of California, Santa Barbara, contributed extensive consultation on issues of experimentation and software engineering.

All authors of this paper are presently engaged in DEDIX-related research activities at UCLA, except as noted next. Udo Voges edited the first draft of this paper prior to returning to his permanent position at the Kernforschungszentrum Karlsruhe, Federal Republic of Germany. Kam Sing Tso has recently assumed a position at the Jet Propulsion Laboratory, Pasadena, California, U.S.A. We also wish to acknowledge the idea of fusing the XEROX Worm and DEDIX concepts, which is due to Nick Lai, a staff member at the UCLA Center for Experimental Computer Science.

References

[Anderson 1981] T. Anderson and P. A. Lee, "Fault Tolerance: Principles and Practice," *Prentice Hall International,* London, England, 1981.

[Anderson 1985] T. Anderson, P. A. Barrett, D. N. Halliwell, D. N. and M. R. Moulding, "An Evaluation of Software Fault Tolerance in a Practical System," *Digest of FTCS-15, the 15th International Symposium on Fault-Tolerant Computing,* Ann Arbor, Michigan, June 1985, pp. 140-145.

[Avižienis 1975] A. Avižienis, "Fault-Tolerant and Fault-Intolerance: Complementary Approaches to Reliable Computing," *Proceedings of the 1975 International Conference on Reliable Software,* Los Angeles, April 1975, pp. 458-464.

[Avižienis 1977] A. Avižienis and L. Chen, "On the Implementation of N-version Programming for Software Fault Tolerance During Execution," *Proceedings of the 1st IEEE-CS International Computer Software and Applications Conference (COMPSAC 77),* Chicago, November 1977, pp. 149-155.

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

[Avižienis 1985a] A. Avižienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software," *Digest of FTCS-15, the 15th International Symposium on Fault-Tolerant Computing,* Ann Arbor, Michigan, June 1985, pp. 126-134.

[Avižienis 1985b] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 12, December 1985, pp. 1491-1501.

[Chen 1978] L. Chen and A. Avižienis, "N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation," *Digest of FTCS-8, the 8th International Symposium on Fault-Tolerant Computing,* Toulouse, France, June 1978, pp. 3-9.

[Cristian 1982] F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers,* Vol. C-31, No. 6, June 1982, pp. 531-540.

[Goguen 1979] J. A. Goguen and J. J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," *Proceedings of the Conference on the Specification of Reliable Software,* Cambridge, MA, April 1979, pp. 170-189.

[Gunningberg 1985] P. Gunningberg and B. Pehrson, "Specification and Verification of a Synchronization Protocol for Comparison of Results," *Digest of FTCS-15, the 15th International Symposium on Fault-Tolerant Computing,* Ann Arbor, Michigan, June 1985, pp. 172-177.

[Guttag 1985] J. V. Guttag, J. J. Horning and J. M. Wing, "Larch in Five Easy Pieces," *Digital Equipment Corporation Systems Research Center, Report No. 5,* Palo Alto, California, July 24, 1985.

[Kelly 1983] J. P. J. Kelly and A. Avižienis, "A Specification-Oriented Multi-Version Software Experiment," *Digest of FTCS-13, the 13th International Symposium on Fault-Tolerant Computing,* Milano, Italy, June 1983, pp. 120-126.

[Kelly 1986] J. P. J. Kelly, A. Avižienis, B. T. Ulery, B. J. Swain, R. T. Lyu, A. Tai and K. S. Tso, "Multi-Version Software Development," *Proceedings of the IFAC Workshop SAFECOMP 86,* Sarlat, France, October 1986, pp. 43-49.

[Kim 1984] K. H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," *Proceedings of the 4th IEEE International Conference on Distributed Computing*
Systems, San Francisco, California, May 1984, pp. 526-532.

[Lamport 1982] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 3, July 1982, pp. 382-401.

[Locasso 1980] R. Locasso, J. Scheid, V. Schorre and P. Eggert, "The Ina Jo Specification Language Reference Manual," *System Development Corp., Tech. Rep. TM-6889/000/01,* Santa Monica, California, November 1980.

[Melliar-Smith 1982] P. M. Melliar-Smith and R. L. Schwartz, "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.

[Popek 1981] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the 8th Symposium on Operating Systems Principles,* Pacific Grove, California, December 1981, pp. 169-177.

[Ramamoorthy 1981] C. V. Ramamoorthy, Y. Mok, F. Bastani, G. Chin and K. Suzuki, "Application of a Methodology for the Development and Validation of Reliable Process Control Software," *IEEE Transactions on Software Engineering,* Vol. SE-7, No. 6, November 1981, pp. 537-555.

[Shoch 1982] J. F. Shoch and J. A. Jupp, "The 'Worm' Programs – Early Experience with a Distributed Computation," *Communications of the ACM,* Vol. 25, No. 3, March 1982, pp. 172-180.

[Tai 1986] A. T. Tai, "A Study of the Application of Formal Specification for Fault-Tolerant Software," *M.S. thesis,* UCLA Computer Science Department, Los Angeles, California, June 1986.

[Tso 1987a] K. S. Tso, "Recovery and Reconfiguration in Multi-Version Software," *Ph.D. dissertation,* UCLA Computer Science Department, University of California, Los Angeles, March 1987; also *Technical Report No. CSD-870013,* March 1987.

[Tso 1987b] K. S. Tso and A. Avižienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation," *Digest of FTCS-17, the 17th International Symposium on Fault-Tolerant Computing,* Pittsburgh, Pennsylvania, July 1987.

[Voges 1982] U. Voges, F. Fetsch and L. Gmeiner, "Use of Microprocessors in a Safety-Oriented Reactor Shut-Down System," *Proceedings EUROCON,* Lyngby, Denmark, June 1982, pp. 493-497.

[Wensley 1978] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak and C. B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE,* Vol. 66, No. 10, October 1978, pp. 1240-1255.

[Zave 1986] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transaction on Software Engineering,* Vol. SE-12, No. 2, February 1986, pp. 312-325.