097144

861-111-107

aerospatiale SOCIÉTÉ NATIONALE INDUSTRIELLE

37, BOULEVARD DE MONTMORENCY, 75781 PARIS CEDEX 16 - TEL. : 524-43-21

861-111-107

A/DET/SG - Doc

PUBLICATIONS 1985
****

NOM DE L'AUTEUR(S) : TRAVERSE P.J.

DATE DE LA CONFERENCE : 19-21 Juin 1985

LIEU DE LA CONFERENCE : University of Michigan
Ann Arbor - Michigan. USA

SOCIETE ORGANISATRICE :
IEEE Computer Society

TITRE DE L'EXPOSE (ou ARTICLE) :
The UCLA-DEDIX system : a distributed testbed for multiple-version software

NATURE ET REFERENCES DE LA PUBLICATION
(conférence proceedings, titre de la revue et date...)
acte de IEEE "FTCS 15" (15th Annual International Symposium on fault-tolerant computing)

OU SE TROUVE LE TEXTE ORIGINAL, AVEC SES REFERENCES :
chez l'auteur (sans référence)

THE UCLA DEDIX SYSTEM: A DISTRIBUTED TESTBED FOR MULTIPLE-VERSION SOFTWARE

A. Avižienis, P. Gunningberg[1], J.P.J. Kelly, 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

To establish a long-term research facility for further experimental investigations of design diversity as a means of achieving fault-tolerant systems, we have designed and implemented the UCLA DEDIX (DEsign DIversity eXperiment) system, a distributed testbed for multiple-version software, at the UCLA Center for Experimental Computer Science. DEDIX is part of the Center's Olympus Net local network, which utilizes the Locus distributed operating system to operate a set of twenty VAX 11/750 computers. DEDIX will be used in second-generation experiments now being designed and coordinated at four universities to measure the efficacy of design diversity and to investigate reliability increases under large-scale, controlled experimental conditions. The DEDIX system is described and its application is discussed in this paper. A review of current research is also presented.

1 Introduction

Originally, fault-tolerant architectures were developed to tolerate *physical* faults that are due to random failure phenomena in the hardware of a computer system. Often, identical hardware channels are used in simultaneous multiple computations in order to attain fault-tolerance. The basic assumption is that the physical faults are uncorrelated. More recently, the tolerance of *design* faults, especially in software, has gained increased attention. Here, it is not possible to use identical copies, since the same fault will manifest itself in all of them. *Design diversity* is the approach in which redundant hardware and software elements are independently designed to meet system requirements. These redundant diverse elements are used in multiple computations in order to tolerate design faults.

Software design diversity, or *N*-version programming [Avi77], is defined as the generation of $N \geq 2$ software "versions" from the same specification. The goal of the specification is to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations. Thereafter, versions are independently written by *N* programming teams that do not interact with respect to the programming process. Since the versions are written independently, it is hypothesized that they are not likely to contain the same errors, i.e., that errors in their results are uncorrelated. In a series of small scale experiments, multiple versions have been executed in parallel and the results from them have been compared and voted. These "0th generation" experiments demonstrated the feasibility of the concept and its effectiveness in dealing with software faults [Chen78]. It was observed that a major complication, compared to voting on identical copies, is that the results might be different due to diversity, but still similar and correct. For example, a floating point algorithm can be written in several ways yielding slightly different results. The decision algorithm must accept these similar results so that a version will not be discarded unnecessarily. This early research also confirmed the practicality of experimental investigation and confirmed the need for high-quality software specifications, since many related errors could be traced to a poor specification.

The principal aim of the subsequent first generation research was the investigation of software specification techniques and the types and causes of software design faults. Improvements both in software specification techniques, and in the use of those techniques, were proposed [Kell83].

Planning for the second generation experiments is now underway. UCLA is cooperating with the University of Illinois, the University of Virginia, and North Carolina State University to conduct large scale experiments under the sponsorship of NASA. Hypotheses on correlated errors have been formulated and will be validated, also the cost effectiveness and the reliability increase will be estimated. To establish a long-term research facility for these second generation experimental investigations, the DEsign DIversity eXperiment system (DEDIX), a distributed testbed at the UCLA Center for Experimental Computer Science has been designed and implemented. This paper describes the requirements of DEDIX, the N-version environment, and the design, implementation, and current experience with DEDIX. Besides serving as an experimental vehicle, DEDIX is available as a node with very high reliability for other users at UCLA.

---

[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

1.1 DEDIX Functional Requirements

The general functional requirements of DEDIX are:

Distribution: the versions should 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 an arbitrary value of N without modifications;

Decision algorithm: a reliable decision algorithm that determines a single decision result from the multiple version results must be provided. The algorithm must be able to tolerate and to treat allowable differences in numerical values and slightly different formats (e.g., misspellings) in human-readable results;

Environment: DEDIX must run on the distributed Locus environment at UCLA 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.

The DEDIX system can in many ways be looked on as an extension of the SIFT system, [Wens78] that is able to tolerate both hardware and software faults. Both have the same type of partitioning, with a decision algorithm at each site that processes broadcast results, and a Global Executive at each site that takes consistent reconfiguration decisions. DEDIX is extended to allow diversity in results and in version execution times. The SIFT system is a clock (frame) synchronous system that uses a clock to predict *when* results should be available for comparison. This synchronization technique does not allow diversity in execution times and unpredictable delays in the communication, which both can be found in a distributed N-version environment. Instead, a synchronization protocol is used in DEDIX, which does *not* use reference to any notion of global time within the system.

1.2 Related Research

A second 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 [Ande81]. 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. This technique is currently being investigated at several locations and DEDIX can support the execution of distributed recovery block programs with relative ease. Several important research activities related to *N*-version programming and recovery block techniques have been reported recently [Ande83, Cris82, Gmei79, Kim84, Rama81, Voge82].

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. They interact with each other and with their environment, i.e., a user, so that together they can be seen as a *system*. DEDIX itself *does not add any functions* to the system. Its purpose is to enhance the reliability of the system and to provide a transparent interface to the users, versions, and input/output system, so that they should not be aware of multiple versions and recovery algorithms. An abstract view of a system with N versions is given in Figure 1. Informally speaking, DEDIX provides the following services:

- it handles requests from the user and distributes them to all active versions;
- it handles requests from the versions to have results corrected, and to distribute corrected results to the versions and to the user;
- it takes decisions on whether or not the results from the versions agree;
- it takes decisions on whether or not to discard faulty versions.

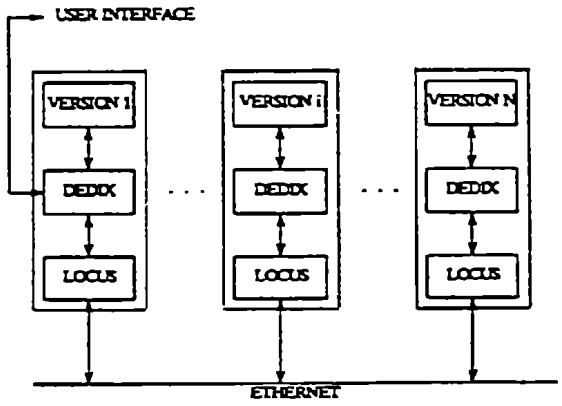

Fig. 1. The N sites of DEDIX

Partitioning of DEDIX. The required services of DEDIX can be mapped either onto a single processor running all versions sequentially, or onto a multiprocessor system, running one local version on each processor. If it is mapped onto a single processor, then the *system* is vulnerable to some hardware and software faults that may cause errors in the operating system or DEDIX software. It is of course possible to use design diversity here as well, but some hardware faults will still cause the single shared processor to fail. Also, a performance penalty is paid if the versions share the same

processor. In a multiprocessor environment, it is possible to partition the system to protect it against most hardware faults as well. This can be done by providing each processor with its own local version, operating system, and decision algorithm. Some interprocessor communication facility must be common to all processors in order to be able to exchange results. It should be noted that the DEDIX design is suitable for any specified number $N \geq 2$ of processors and versions.

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 correct 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 *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 DEDIX software, etc. A missing result at a site might also be caused by an excessive communication delay. The result was produced but never reached the other sites. It is possible to identify why it is missing. When it is excessively delayed, the particular sending site will detect the discrepancy between what it sent and what the other sites observed.

Time-out Function. The only way to detect that a version did not produce a result when it was expected to or when the result is "stuck" somewhere in the communication system is to use a time-out function, i.e., to require that a version must produce a result within a time-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 piece of computation and all versions must produce results within this time interval 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 computations.

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-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.

Both techniques can exist together in DEDIX, and it might depend on the application, input/output, the computing environment, and real time requirements, which type is chosen. Both techniques require that redundant computations should start almost at the same time at each processor and that redundant user input also must arrive within the time interval. In the current implementation, the latter technique is implemented, due to the implementation environment and 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.

Layered Design. The layered design of DEDIX has many advantages. One of the most important is that it reduces complexity. The purpose of each layer is to offer certain *services* to the higher layers, shielding the higher layers from details on how the offered services actually are implemented. Each layer adds new services to those provided by the lower layers. The structures and algorithms of one layer are not visible outside that layer. For example, a layer can provide a fault-tolerant service that includes redundancy and algorithms for fault-detection and recovery. Another important advantage is that the implementation of a given layer can be changed without affecting the other layers, provided the service of the layer is unchanged.

DEDIX is designed as a set of hierarchically structured layers to reduce complexity. Each site has an identical set of layers and entities. These layers, from the bottom to the top are: *Transport Layer, Synchronization Layer, Decision and Executive Layer, and Version Layer.* Each layer provides a set of services, which are described below and shown in Figure 2.

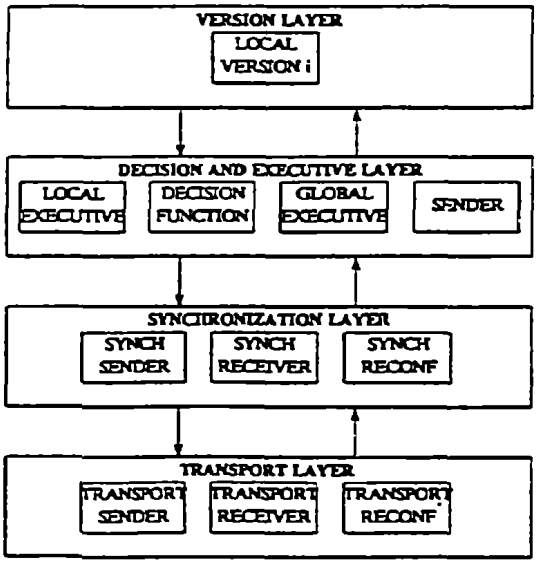

Fig. 2. The layers at site i of DEDIX.

2.2 The Transport Layer

Purpose: 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.

Comments: Since there is no such thing as a fault-free connection, the Transport Layer must be identified with the likelihood that a message is lost or damaged, i.e., the reliability of the service must be stated. Also of interest for the higher layers are its response time and throughput. The Transport Layer is expected to use a redundant underlying communication structure to meet the reliability requirements.

Implementation: Currently, a single ring structure of inter processor UNIX pipes is used. Since this implementation does not allow a site crash, a redundant interconnection structure is under implementation. The initial ring implementation provided us with some determinism in the system which made it much easier to observe and debug the Transport Layer.

2.3 The Synchronization Layer

Purpose: For each physically distributed site, this layer broadcasts results (using the Transport service) and collects messages with the results, ("cc-vector") from all other sites. The layer only accepts results that are both broadcast within a certain time interval and that will arrive within the same time interval. The collected results are delivered to the Decision function. The layer accepts a new set of results when every site has confirmed that all or enough of the previous results have been delivered.

Comments: The processors need to be event-synchronized in order to ensure that results from corresponding computations are compared. Otherwise, if two sets of results from two different computations are compared, the Decision algorithm might wrongly conclude that some of the processors are faulty. Traditionally, this synchronization has been obtained by referring to a common clock or set of clocks. The SIFT system [Mell82] 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 underlying distributed system and the versions have the following characteristics which make the clock synchronous technique difficult to use or impractical in DEDIX:

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

Implementation: A synchronization protocol is designed to provide the service. It ensures that the results that are compared by the Decision function are from the same cross-check (cc) point in each version. The versions are stopped 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 and to allow slow versions to catch up, the previously mentioned time-out technique is used by the protocol.

Using this protocol means that the synchronization of the system is based on:

- the fact that correctly working versions must produce exactly the same number of cc-vectors;
- that correctly working versions have similar execution times, i.e., they will produce results within or before a specified time-out interval;
- that "missing" or disagreeing results do not exist at a majority of sites.

Each site has both a Sender and a Receiver entity in this layer, which communicate with the other site's corresponding entities 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 acknowledgements 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 acknowledgements, it will indicate this to the Decision and Executive layer. This indication is used to restart the versions. It might seem unnecessary to use acknowledgements, since a Receiver can inform the Sender that it has received enough results. However, the Receivers might have an inconsistent view on the number of received results. For example, in a three site environment, two sites might get three results immediately while the third site only gets two. The third site cannot yet accept a new set of results. By using this indication, it is ensured that *all* Receivers are ready to accept a new set of results. The specification and verification of the protocol is described in a companion paper [Gunn85].

2.4 The Decision and Executive Layer

Purpose: This layer receives results (specified as "cross-check vectors") from its local version, takes agreement decisions on results received from all other versions and delivered by the Synchronization layer, determines whether the local version is faulty or not, and takes recovery decisions. Corrected results are forwarded to the local version. It controls the input/output of the local version. All exceptions that cannot be handled elsewhere are directed to this layer.

Implementation: The layer has four entities, a Sender, a Decision function, and two entities for controlling the recovery process, a Local Executive and a Global Executive. The Sender entity receives the *requests* from the local version and it responds to the version when a decision has been taken with corrected results. There are four different types of *normal* comparison requests: *intermediate state* cc-vector, *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. At an *input* request, a decision is first taken on the format vector before the actual input read is performed. When the local version has raised an exception from which it cannot recover, it will use the *abnormal exception request*, which is immediately directed to the Local Executive.

The Sender entity attaches an *occurrence number* to the current cc-identifier (cc-id) of the cc-vector. The occurrence number is used to uniquely identify a cc-id, since the same cc-id will appear in loops and other repeated program sequences. For each time a version is requesting a decision, the occurrence number for that cc-id request is incremented.

Input/Output System. The input/output system to the versions is designed to be replicated as well. However, in the current implementation a centralized terminal connection is used for all input/output, and the data to be printed or read is distributed to all versions. They will view the data as replicated. The interface between DEDIX and the input/output system is similar to the interface between DEDIX and the local version. For example, a read from a terminal might be timed-out if it does not respond in phase with the other terminals, and the output data is run through the Decision function before the actual output. The request to read data is also run through the Decision function to ensure consistency.

The Global Executive. The *purpose* of the Global Executive is to: a) collect error reports from the Decision function and the Local Executive, b) exchange error reports with every other active Global Executive, and c) decide on a new reconfiguration, based on all error reports. The current implementation is rudimentary. The functions of the Global Executive are basically the same as in the SIFT system. One difference is that the SIFT executive is invoked at predefined time intervals at all sites. This is not possible in the DEDIX system, since the sites might have a different state of computation at the same time. Instead, the Global Executive is invoked after a preset number of exchanges of results (= number of decisions) has taken place. The number of exchanges *is the only consistent computation state* in all sites. That is, by referring to this number, it is possible to ensure that all correctly working sites will exchange error reports and decide on a reconfiguration at the same state of computation. This number is kept consistent by the synchronization protocol.

Error Reports. Every local Executive has an error report table, with one entry for each site. In the current implementation this entry is an error counter for that site. The Local Executive increments the counter for a site, each time that the site has either a disagreeing or missing result. This means that the Local Executive 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 slightly different error reports.

The control is moved to the Global Executive when an exchange of results should take place. The Global Executive at a site will temporarily take the place of the local version and use the broadcasting and decision functions of the underlying layers. The Error Report is put into a regular message and delivered to the Synchronization layer, which does not perceive the difference. The Synchronization layer collects messages as usual and they are run through the Decision function in order to ensure that every site has a consistent view on the error reports.

The Reconfiguration Decision. The Global Executive will get a consistent error report on which it decides on the reconfiguration. In the current implementation, only a degradation can be done. It is not possible to start an inactive site. A site is proposed to be disconnected if the number in the error reports counter exceeds a predefined threshold value, say 50% of all exchanges. All Global Executives propose a new configuration that is also broadcast to every other site and run through the Decision function. The proposed configurations are voted on bit-by-bit which will ensure a consistent view on a new configuration at every correctly working site.

A degradation means that the Local Executive instructs the receiving entities to stop listen to that site, or if the faulty version is local to the same site, to terminate the version and to quit sending messages. The new number of expected results is adjusted accordingly. After a site is degraded, it will still collect messages and operate input/output, but it will not deliver them to the local version, provided that the fault only affects the version.

The Local Executive. The Local Executive is activated when the Decision function indicates that the result is not unanimous, or when some unrecoverable exception is signaled from the local version or some other layer. The Local Executive will first try to recover locally from the fault before it either reports the problem to the Global Executive or, if it is considered as fatal to the site, closes 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 they are independent of the implementation. Among them are the raised exceptions from an unanimous result, when a communication link is disconnected, and 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, like segmentation faults, process termination, invalid system call, etc., belong to this class. Other examples are all the exceptions defined in DEDIX, like signaling when a function is called with an invalid parameter, or when an inconsistent state exists. Most of these exceptions will force an orderly close down in order to be able to provide

data for analysis.

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

2.5 The Version Layer

The purpose of this layer is to interface the i-th (local) version with the DEDIX system and to correct the state of the variables that are incorrect according to the Decision function. The function doing the interfacing is called the Cross-Check, or CC-Function since it is called as a function to 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 internal representation of a cc-vector in DEDIX is hidden to the version program. The CC-Function writes back the corrected results into the version.

2.6 The Decision Algorithm

The Decision Algorithm is used to determine a single decision result from the N-version results. The Decision Algorithm 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 the case that a decision result cannot be determined, a higher level recovery procedure may be invoked.

In DEDIX we have implemented a generic Decision Algorithm which may be replaced by user written routines provided that the interfaces are preserved. This allows application-specific decision algorithms to be incorporated in those cases where the default mechanisms are inappropriate; for example, this may occur because of lack of sensitivity, or unnecessary elimination of program versions.

The generic Decision Algorithm is hierarchical in nature. The algorithm attempts to determine a decision by applying the following major decision classes sequentially:

(1) bit by bit - identical match only;

(2) cosmetic - detecting character string differences caused by misspelling or character substitution;

(3) numeric - integer and real number decisions.

All numeric decisions use a median value and it can be proved that, so long as the majority of versions are not faulty, the median of all responses is acceptably close to a supposed ideal value. Numeric values are allowed to be different within some "skew interval" thus allowing results to be non-identical but still similar.

2.7 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.

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 stimuli 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.

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.

Statistic 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.

3 Experimental Goals of the DEDIX Testbed

The second generation experiments at UCLA have two fundamental goals: the investigation and evaluation of various fault-tolerance mechanisms and the analysis and characterization of the fault distributions of highly reliable program versions.

3.1 Fault Tolerance Mechanisms

We expect to obtain quantitative experimental results about the effectiveness of the fault tolerance mechanisms. We also plan to evaluate the possible loss of performance due to the operation of the fault-tolerance mechanisms in the absence of faults, as well as the cost of error recovery.

A problem area that is being thoroughly examined is the recovery of failed versions through backward and forward recovery, and reinitialization. Since we assume that all versions are likely to contain design faults, it is critical to be able to recover these versions as they fail, rather than merely degrade to N-1 versions, then N-2 versions and so on. A pilot experiment is underway in which failed versions are recovered without requiring the specification of the entire internal state.

An important and interesting application area that often requires very high reliability and availability is real-time execution of time-critical applications. However, the current implementation using Locus is likely to be too slow for this purpose. Despite this limitation of Locus, its functional architecture 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.

We will investigate the effectiveness of design diversity as a means of increasing software reliability within a constrained budget. We are interested in the costs of removing bugs and of enhancement. By combining relatively unverified, unvalidated software versions to produce highly reliable multi-version software we may be able to decrease cost while increasing reliability. Most errors in the software versions will be detected by the Decision Algorithm during on-line productive use of the system. The software faults then can be fixed while limiting their effects on system availability.

Enhancing multiple software versions is more difficult. Specifications should be sufficiently modular and structured so that enhancement will generally affect few modules. The extent to which each module is affected can then be used to determine whether (1) existing versions should be modified to reflect the enhancement, (2) existing versions should be discarded and new versions produced, or (3) new versions should be produced to implement the enhancements and old versions kept to implement the original requirements. Experiments will be conducted to gain insights into the criteria to be used for a choice.

In the "mail-order" concept members of fault tolerance research groups at several universities will write software versions for use in a large experiment. We expect that software versions produced at geographically separate locations, by people with different experience who use different programming languages, will contain substantial design diversity. It may be possible to utilize the rapidly growing population of computer hobbyists on a contractual basis to provide individual module versions at their own locations. This would not require a large concentration of skilled people and would allow for the loss of individual programmers.

3.2 The Fault Distributions of Highly Reliable Versions

The other major goal of the second generation experiments concerns the distribution of faults in highly reliable program versions. A recent theoretical analysis of redundant software has claimed that there are major differences between the models needed to describe redundant software faults and independent hardware faults [Eckh85]. Indeed, a clear need was seen for empirical data to truly assess the effects of errors on highly reliable software systems.

A model experiment has been specified in which 15 general guidelines and 10 specific tasks are identified [Kell82], and second-generation experimentation is now underway at four universities (UCLA, University of Virginia, University of Illinois and North Carolina State University) [Avi84] to measure the efficacy of design diversity and to demonstrate potential reliability increases under large-scale, controlled experimental conditions.

We intend to produce these software versions under controlled conditions that approximate the development methodologies and environments used by advanced industrial facilities. We will conduct extensive logging of work periods and events such as error discovery, specification questions and answers, and test suite execution. The experimenters will provide a complete high-level external specification. At all stages, questions about the specifications will be submitted by electronic mail, reviewed by the experimenters, and responded to by electronic mail. The determination that a question reveals a flaw in the specifications will cause a change to be broadcast to all programmers at all sites. The deliverable items will 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 acceptance test. We will study fault distributions by conducting extensive testing of the versions with randomly generated test data. The nature and cause of all detected errors will be analyzed.

4 Specification Issues

Significant progress has occurred in the development of formal specification languages since our previous experiments [Avi84]. Our current goal is to compare and assess the applicability to practical use by application programmers the following formal program specification methods:

(1) The CLEAR specification language developed at Edinburgh University and SRI International; [Burs81]

(2) The LARCH family of specification languages developed at Xerox Palo Alto Research Center and at M.I.T.; [Gutt83]

(3) The OBJ specification language developed at UCLA; [Gogu79]

(4) The Ina Jo specification language developed at SDC; [Loca80]

(5) The "M" specification language, descended from "Z"; [Meye84]

(6) The applicability of Concurrent Prolog as a method of formal specification.

The study focuses on the assessment of the following aspects of the specification languages: (1) The purpose and scope (problem domain); (2) Completeness of development; (3) Quality and extent of documentation; (4) Existence of support environments; (5) Executability and suitability for rapid prototyping; (6) Provision of notation to express timing constraints and concurrency; (7) Methods of specification for exception handling; (8) Extensibility for the specification of fault-tolerant multi-version software.

The outcome of the study will be 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 verification of the 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. The specification will provide an executable prototype of the DEDIX supervisory operating system as well as the application versions. 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 traceable to specification faults and to increase the dependability of the specifications.

5 Conclusion

This paper has presented an overview of a major effort to develop a research environment for software design diversity research at UCLA. The complete DEDIX prototype has been implemented, and second-generation experiments are underway. Several other research efforts also have been initiated.

6 Acknowledgement

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.

References

[Ande81] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, London, England: Prentice Hall International, 1981.

[Ande83] T. Anderson and J.C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983, pp. 355-364.

[Avi77] A. Avižienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution," in *Proceedings COMPSAC 77*, 1977, pp. 149-155.

[Avi84] A. Avižienis and J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol. 17, No. 8, August 1984, pp. 67-80.

[Burs81] R.M. Burstall and J.A. Goguen, "An Informal Introduction to Specifications Using CLEAR," in *The Correctness Problem in Computer Science*, R. Boyer and H. Moore, Ed. New York: Academic Press, 1981, pp. 185-213.

[Chen78] L. Chen and A. Avižienis, "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.

[Cris82] F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers*, Vol. C-31, No. 6, June 1982, pp. 531-540.

[Eckh85] D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Redundant Software Subject to Coincident Errors," NASA, Hampton, Virginia, Tech. Rep. 86369, January 1985.

[Gmei79] L. Gmeiner and U. Voges, "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.

[Gogu79] J.A. Goguen and J.J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," in *Proceedings Specifications Reliable Software Technology*, Cambridge, Mass.: 1979, pp. 170-189.

[Gunn85] P. Gunningberg and B. Pehrson, "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.

[Gutt83] J.V. Guttag and J.J. Horning, "An Introduction to the Larch Shared Language," in *Proceedings IFIP Congress 83*, 1983, pp. 809-814.

[Kell82] J.P.J. Kelly, "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.

[Kell83] J.P.J. Kelly and A. Avižienis, "A Specification Oriented Multi-Version Software Experiment," in *Proceedings 13th IEEE International Symposium on Fault-Tolerant Computing Systems*, Milan, Italy: June 1983, pp. 121-126.

[Kim84] K.H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," in *Proceedings IEEE 4th International Conference on Distributed Computing Systems*, San Francisco, California: May 1984, pp. 526-532.

[Loca80] R. Locasso, J. Scheid, V. Schorre, and P. Eggert, "The Ina Jo Specification Language Reference Manual," System Development Corp., Santa Monica, CA, Tech. Rep. TM-6889/000/01, November, 1980.

[Mell82] 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. 616-234.

[Meye84] B. Meyer, "A System Description Method," in *International Workshop on Models and Languages for Software Specification and Design*, B.G. Babb II A. Mili, Ed. Orlando, Fla.: March 1984, pp. 42-46.

[Rama81] C.V. Ramamoorphy and *et al.*, "Application of a Methodology for the Development and Validation of Reliable Process Control Software," *IEEE Trans. Soft. Eng.*, Vol. SE-7, No. 6, November 1981, pp. 537-555.

[Voge82] U. Voges, F. Fetsch, and L. Gmeiner, "Use of Microprocessors in a Safety-Oriented Reactor Shut-Down System," in *Proceedings EUROCON*, Lyngby, Denmark: June 1982, pp. 493-497.

[Wens78] 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.