Tolerance of design faults in software and in hardware is the challenge of the eighties. How can we meet the challenge? Design diversity offers a solution.

Fault Tolerance by Design Diversity: Concepts and Experiments

Algirdas Avižienis and John P. J. Kelly
UCLA Computer Science Department

Fault tolerance is the survival attribute of computer architectures; when a system is able to recover automatically from fault-caused errors, and to eliminate faults without suffering an externally perceivable failure, the system is said to be fault tolerant.[1] Fault tolerance is supported by both hardware and software elements of the system, including redundant resources and detailed implementations of fault detection and recovery algorithms. Originally, fault-tolerant *architectures* were developed to tolerate *physical* faults that occur because of random failure phenomena in the hardware of a system. More recently, the tolerance of *design* faults, especially in software, has been added to the objectives of fault tolerance.

The use of redundant copies of hardware, data, and programs has proven to be quite effective in the detection of physical faults and in subsequent system recovery. However, design faults—which are introduced by human mistakes or defective design tools—are reproduced when redundant copies are made; such replication of faulty hardware or software elements fails to enhance the fault tolerance of the system with respect to design faults.

Design diversity is the approach in which the hardware and software elements that are to be used for multiple computations are not copies, but are independently designed to meet a system's requirements. Different designers and design tools are employed in each effort, and commonalities are systematically avoided. The obvious advantage of design diversity is that reliable computing does not require the complete absence of design faults, but only that those faults not produce similar errors in a majority of the designs. This and other advantages, as well as some limitations of design diversity, are discussed in this article.

Implementation and use of design diversity

The multiple-computation approach to fault tolerance. A very effective approach to the implementation of fault tolerance has been the execution of multiple (N-fold, with $N \geq 2$) computations with the same objective: a set of results, derived from a given set of initial conditions and inputs. Two fundamental requirements apply to such multiple computations:

(1) the consistency of initial conditions and inputs for all N computations must be assured; and
(2) a reliable decision algorithm that determines a single *decision result* from the multiple results must be provided.

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. It is also possible that the decision result cannot be determined, in which case a higher level recovery procedure may be invoked. The decision algorithm, itself, is often implemented N times—once for each computation in which the decision result is used—in which case only one computation is affected by the failure of any particular decision element (such as the majority voter).

Multiple computations have been implemented by N-fold ($N \geq 2$) replications in three domains: *time* (repetition), *space* (hardware), and *program* (software). The additional attribute of diversity will be addressed later. First, we must define and discuss a shorthand notation to describe the various possible approaches to multiple computation.

0018-9162/84/0800-0067$01.00

The reference, or *simplex*, case is that of one execution (simplex time $1T$) of one program (simplex software $1S$) on one hardware channel (simplex hardware $1H$), described by the notation: $1T/1H/1S$. For simplex time ($1T$), multiple computation cases consist of concurrent execution of N copies of a program on N hardware channels ($1T/NH/NS$). Examples of such systems are NASA's Space Shuttle ($N=4$), SRI's SIFT System ($N\geq 3$), Draper Laboratory's Fault-Tolerant Multiprocessor ($N=3$), and AT&T Bell Laboratories No. 1 ESS ($N=2$) computers.[1]

In the N-fold time cases (NT), sequential executions of a computation are performed. Some transient faults are detectable by the repeated ($N=2$) execution of the same copy of a program on the same machine ($2T/1H/1S$). A common practice in coping with system crashes has been the loading and running of a new copy of the program on the same hardware ($2T/1H/2S$). A more sophisticated fault-tolerance approach that is used in Tandem computers, for example, is the execution of a copy of the program on a standby duplicate channel ($2T/2H/2S$). The repetition of the same copy of a program on a duplicate channel ($2T/2H/1S$) is another possible variation.

The types of faults in multiple computations. The usual partitioning of faults into "single-fault" and "multiple-fault" classes needs to be refined when we consider multiple computations. Faults that affect only one in a set of N computations are called simplex faults. A simplex fault does not affect other computations, although it may be either a single or a multiple fault within one channel. Simplex faults are very effectively tolerated by the multiple-computation approach as long as input consistency and an adequate decision algorithm are provided.

Faults that effect M out of the N separate computations are called *M-plex* faults ($2\leq M\leq N$); they affect computations that form a set which is used to arrive at a decision result. Typically, multiple occurrences of faults are di-

Is it a fault, an error, or a failure?

Consistent and well-defined meanings need to be attached to the basic terms *fault, error,* and *failure* when fault tolerance is discussed.* We observe the following situation: a system named resource *(r)* is delivering an expected service *(s)* to a system or a person named user *(u)*.

Failures. A *failure* occurs when the user perceives that the resource ceases to deliver the expected service. Examples of failures are

(1) the reservation agent *(u)* perceives that the computer *(r)* acknowledges an inquiry but does not produce any response *(s)*;
(2) the "watchdog" timer *(u)* perceives that the currently executing program *(r)* has not reset *(s)* the timer before it reaches the zero value;
(3) the CPU *(u)* perceives that the memory *(r)* has delivered a word *(s)* with the wrong parity; and
(4) transistor *B (u)* perceives that the output of transistor *A (r)* does not change *(s)* after a test input is applied to *A* by *B*.

The user may choose to identify several types of failures, such as minor, severe, or catastrophic. More than one level of service may be defined, with some failures only lowering the quality of service.

Errors. An *error* occurs when some part of the resource assumes an undesired state. Such a state is contrary to the specification of the resource or the expectation (requirement) of the user. Examples of errors are

Parity error. All words are stored in a memory with odd parity, but the "read" operation delivers a word that has even parity;
Comparison error. Two identical adders receive the same operands and simultaneously deliver sums to a comparator that are not identical in every bit position;
Timing error. The two adders (just mentioned) simultaneously receive the same operands, but only one adder delivers the sum to the comparator at the predetermined time;
Program error. The identical outputs of both adders contain overflow that should have been prevented by operand range checking earlier in the program.

When an error is perceived at the boundary of the resource, the user considers it to be either a failure, or an acceptable (although less desired) level of service. For example, the balance of a checking account that is repeatedly delivered with a parity error is perceived as a failure. On the other hand, one unchanging pixel on a display screen is an error that is usually perceived not as a failure, but only as a nuisance. An error is *latent* as long as it is not detected by a detection algorithm and does not cause a failure.

Faults. A *fault* is detected when either a failure of the resource occurs, or an error is observed within the resource. The cause of the failure or error is said to be a fault. In most cases the fault can be identified; in some it remains a hypothesis that cannot be adequately verified. Examples of faults are

A permanent physical fault. The output of an AND gate is stuck on logic one;
A transient physical fault. An alpha particle impact changes the state from one to zero in a dynamic MOSFET memory cell;

vided into two classes: *independent* and *related*. We say that related M-plex faults are those for which a common cause, which affects M computations, exists or is hypothesized. Examples of common physical causes include interchannel shorts or "sneak paths," power-supply interruptions, and bursts of radiation. Their effects on the individual computations are not necessarily alike. Another class of related M-plex faults is quite different: it arises from human mistakes made during the design or the subsequent design modification of a hardware or software element. All N computations that use copies of that hardware or software element are ultimately affected in the same way by the design fault when a certain state of the element is reached.

Because of their differing consequences, related M-plex faults are likewise divided into two classes: *design faults* and *operational faults*. Design faults are generic to an entire set or family of hardware/software copies, since they result from a deviation of the design from its specification. All other related M-plex faults are operational. Their relationship is not traceable to a design mistake, but rather to the physical, logical, or temporal proximity of the affected computations.

The consequences of faults are *errors* that can be observed at points at which either single-computation fault-detection algorithms or N-computation decision algorithms are applied to the results. The M simplex errors that are caused by related M-plex faults may be either *identical* or *distinct* at the points of observation. Operational M-plex faults may produce errors of either type, although they are much more likely to cause distinct errors. Design faults will produce identical errors in all computations if identical copies of hardware and software elements are employed to execute multiple computations that have the same inputs and initial values. Such total susceptibility to design faults is the most serious limitation of the multiple-computation approach.

A design fault. A program does not properly check input data for range constraints; and
A specification fault. The range constraints for input data are ambiguously described in the specification.

A fault is *latent* as long as it has not caused any errors, but exists in the resource as a potential cause. For example, a stuck-on-one gate in an unpowered spare CPU is a latent fault.

What makes the difference? For a resource of any size and form—whether it be a computer, program, memory (systems/subsystems), or even a single transistor—we say that

a *failure* is loss of service that is perceived by the user at the boundary of the resource (that is, the point at which the resource is monitored by the user—for example, this can be a screen monitor for software systems or it can be read-outs for actual circuitry for hardware systems);
an *error* is an undesired resource state that exists either at the boundary or at an internal point in the resource and may be perceived as a failure when it is propagated to and manifested at the boundary; and
a *fault* is the identified or hypothesized cause of the error or failure.

The difference between a failure, an error, or a fault is determined by the location of the service boundary of the resource. Loss of service to the user at the boundary is perceived as a failure; an undesired state within the resource, caused by a fault, is considered an error. Since resources are nested, the fault, itself, may be perceived as a failure when the service boundary is moved inward and defined to be located "at the fault."

For example, consider a bank teller *(u)* who requests the computer *(r)* to deliver the balance *(s)* in a checking account. The computer response "reading error" indicates that several attempts to read the balance were unsuccessful. The computer *(r)* has failed to deliver the balance *(s)*. Inside the computer, the parity checker on the memory output bus records a parity error after each read operation; the computer gives up after several tries and sends the message of "reading error." The cause of the persistent parity error, that is, the fault, is hypothesized to be a "stuck" memory cell.

Now consider storing the test word "all ones" in the affected memory location. The service boundary—the point at which the location can be monitored—is now the output of a memory cell. A read-out of "zero" from one memory cell is a failure, since the cell has not reproduced the input. An error has occurred (the undesired state change from 1 to 0) due to a fault (the physical breakdown of a semiconductor junction in the cell circuitry).

In conclusion, we note that a fault-tolerant resource detects faults by means of *detection algorithms* that recognize errors. The *recovery* algorithm of a resource carries out the correction (or deletion) of errors and the elimination of faults, if they are permanent. The presence of faults within the resource does not lead to failures at its boundary; that is, the expected service is delivered to the user without the need for preserving perfection in the resource.

*This review reflects the authors' view of concepts that evolved from a continuing discussion conducted by members of IFIP Working Group 10.4, *Reliable Computing and Fault-Tolerance*. Special thanks go to Jean-Claude Laprie for his insistence on the service concept and to Brian Randell for his persuasive views on the recursive nesting of resources.

Design diversity in multiple computations. In N-fold implementations, an effective method of avoiding identical errors caused by design faults is to use independently designed software and/or hardware elements instead of identical copies. This approach applies directly to the parallel, simplex time system $1T/NH/NS$, which can be converted to (1) $1T/NH/NDS$ (where NDS stands for *N-fold diverse software,* which is used in N-version programming,[2,3] described below); (2) $1T/NDH/NS$ (where NDH stands for N-fold diverse hardware); and (3) $1T/NDH/NDS$.

The sequential NT systems have been implemented as recovery blocks[3] with N sequentially applicable alternate programs ($NT/1H/NDS$) that use the same hardware. An acceptance test is performed for fault detection, and the decision algorithm selects the first set of results that pass the test.

A number of variations and combinations of approaches can be readily identified, in addition to the recovery-block and N-version programming methods. First, the acceptance tests on individual versions can be used to support the decision algorithms in NVP. A consistent method for software exception handling has been constructed.[4] Second, fault-detection algorithms in individual hardware channels are applicable in order to distinguish physical faults from design fault.

Third, the recovery-block approach can concurrently use two (or more) hardware channels, either copied, or diverse: $(N/2)T/2H/NDS$ or $(N/2)T/2DH/NDS$. A decision algorithm applied here can support the acceptance tests, since two sets of results are made available in parallel, and real-time constraints are met as long as one channel remains acceptable.

The most general diverse system is, however, $XT/YDH/ZDS$ with acceptance tests and detection of physical faults in each channel supporting the decision algorithm. Other systems result when some features are deleted or diversity is reduced.

Looking back at earlier machines, we may consider that "marginal testing" as used in first-generation hardware was a primitive method of imposing sequential hardware diversity on a single channel ($2T/2DH/1S$), and that repetition of a computation with a different clock rate may be thought of as a form of "time diversity": $2DT/1H/1S$.

The use of design diversity introduces "similarity" considerations about results and errors. First, the decision algorithm may have to determine the decision result from a set of similar, but not identical, results. *Similar* results are defined as two or more results (good or erroneous) that are within the range of variation allowed by the decision algorithm; they are therefore used to determine the decision result. Similar results often differ within a certain range when different algorithms are used in diverse designs. Consequently, similar, but not identical errors can cause the decision algorithm to arrive at an erroneous decision result. The errors are now classified as either *distinct* or *similar* (*identical* errors form a subset of *similar errors*). Finally we note that a new type of M-plex design faults—*independent* M-plex design faults—can cause similar errors and lead to an erroneous decision result.

Eliminating related design faults. To reduce the likelihood of related design faults in a set of computations, it is necessary to employ (1) independent design and implementation techniques such as diverse algorithms, programming languages, translators, design automation tools, and machine languages; and (2) independent (noninteracting) programmers or designers (preferably with a variety of training and experience).

The third and probably the most critical condition is the existence of a complete and accurate *initial formulation* of the requirements that the diverse designs are to meet. This is the "hard core" of this fault-tolerance approach. Latent defects—such as inconsistencies and ambiguities in, or omissions from, the initial formulation—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 formulation is the use of formal, very high level specification languages. Such specifications are executable and can be automatically tested for latent defects. Here, perfection is required only at the highest level of specification; the rest of the design and implementation process—as well as its tools—are required to be only as fault-free as existing resource constraints and time limits permit. The independent generation and subsequent comparison of two specifications, using two formal languages, is the next step toward increasing the dependability of current specifications.

Potential advantages of design diversity. The most immediate and direct opportunity to apply design diversity is in multichannel systems, such as SIFT,[5] that have very complete tolerance of physical faults. The hardware resources and architectural features that can support software diversity are already present, and their implementation of diversity is an extension of existing physical fault-tolerance mechanisms. Hardware diversity can be introduced by choosing, for each channel, functionally compatible hardware building blocks from different suppliers.

A more speculative—and also more general—application of design diversity is its use as a partial substitute for current software verification and validation procedures. Instead of extensive V&V of a single program, two independent versions can be executed in an operational environment, completing V&V concurrently with productive operation. The doubled cost of producing the software is compensated for by a reduction of the V&V time. In some situations, there may also be a decrease in the amount —and therefore the cost—in man hours and software tools needed for the very thorough V&V effort.

Design faults in the V&V software are also less critical. The user can take the less efficient ("back-up") version off line when adequate reliability of operation is reached, and then bring it back on line for special operating conditions that require greater reliability assurance, especially after modifications or after maintenance. A potential system-lifetime cost reduction exists because a duplex, diverse system can support continued operation after latent design faults are uncovered, providing nearly 100 percent availability. The cost of fault analysis and elimination should also be reduced because repairs are less urgent.

The possible use of a "mail order" approach to the production of two or more versions of software modules suggests a very intriguing long-range benefit of the design-diversity approach in software. Programmers, working at their own preferred times and locations and using their own personal computing equipment, can write the versions of software if they are given a formal specification that includes a set of acceptance tests. Two potential advantages of this approach are

- The overhead cost that accrues in highly controlled professional programming environments would be drastically reduced, since this approach allows the programmers free play to work on their own initiative and utilize low-cost home facilities.
- The potential of the rapidly growing number of computer hobbyists to serve as productive programmers would be tapped. For various reasons, many individuals with programming talents cannot fill the professional programmer's role as defined by today's rigorous approaches to quality control and use of centralized sites during the programming process.

Finally, an important reliability and availability advantage to be gained through design diversity may be expected for systems that use VLSI circuits. The growing complexity of VLSI circuits—with 500,000 gates/chip now available and one million gates/chip predicted for the near future—raises the probability of design faults, since a complete verification of the design is very difficult to attain. Furthermore, the design automation and verification tools are themselves subject to undiscovered design faults. Even with multichannel fault-tolerant system designs, a single design fault may require the replacement of all chips of that type, since on-chip modifications are impractical. Such a replacement is a costly and time-consuming process. However, use of design diversity of VLSI circuits does allow the continued use of chips with design faults, so long as their errors are not similar at the circuit boundaries where the decision algorithm is applied. Reliable operation throughout the lifetime of a system can be obtained even if none of the chips has a perfect design and none of the basic structure of the VLSI circuits has been modified.

Development of multiversion software: an experimental approach

Because of the potential advantages of design diversity and design fault tolerance, further studies have been conducted to determine their usefulness as alternatives to V&V, testing, and proof methodology, which aim at delivering fault-free software products and hardware circuits. A research effort at UCLA (started in 1976) was founded on 14 years of continuous investigation into tolerance of physical faults; the goal was to determine the feasibility of adapting N-fold modular redundancy (with majority voting) to provide fault tolerance of software design faults. The first experimental study of this approach, called "N-Version Programming," was completed in 1978. However, suggestions that this approach might be a viable method of software fault tolerance had been published previously.[6] In fact, Dionysius Lardner first made the suggestion in his article "Babbage's Calculating Engine," published in the *Edinburgh Review* in July 1834:

> The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods.[7]

A second approach, already under investigation in 1975, 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. The objective of the RB technique is to perform software design fault detection during runtime by an acceptance test performed on the results of one version. If the test fails, an alternate path of execution is taken to implement recovery. This technique is currently being investigated at several locations, and several important research activities related to NVP and RB have been reported recently.[3,4,8-10]

Initial studies of N-version programming. N-version programming is defined as the independent generation of $N \geq 2$ software modules, called "versions," from the same initial specification.[2] "Independent generation" means that programming efforts are carried out by individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms, programming languages, and translators are used in each effort.

The goal of the initial specification is to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the N programming efforts. It also states all the special features that are needed to execute the set of N versions in a fault-tolerant manner.[6] An initial specification defines (1) the function to be implemented by an N-version software unit; (2) data formats for the special mechanisms: decision vectors ("d-vectors"), decision status indicators ("ds-indicators"), and synchronization mechanisms; (3) the cross-check points ("cc-points") for d-vector generation; (4) the decision algorithm; and (5) the responses to the possible outcomes of decisions. We note that "decision" is used as a general term to mean the determination of a decision result from multiple versions. The term "comparison" usually refers to the $N=2$ case, and "voting" to a majority decision with $N>2$. The decision algorithm explicitly states the range of variation, if such a range exists, that is allowed in determining whether version results are similar.

It is the fundamental assumption of the N-version approach that the independence of programming efforts will greatly reduce the probability that software design faults will cause similar errors to occur in two or more versions. Together with a reasonable choice of d-vectors and cc-points, the independence of design faults is expected to make N-version programming an effective method of achieving design fault tolerance. The effectiveness of the

entire approach depends on the validity of the assumption, and an experimental investigation, deemed essential, was undertaken at UCLA.

The initial research effort at UCLA addressed two questions: (1) What is required in terms of formal specifications, choices of problems, and the nature of algorithms, timing constraints, and decision algorithms to make N-version programming feasible, regardless of the cost? (2) What methods should be used to compare the cost and the effectiveness of the N-version programming approach to the two alternatives: single-version programming and the recovery block[3] approach?

The scarcity of previous results and an absence of formal theories on N-version programming 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 generate a set of programs. Once generated, the programs were executed in a simulated multiple-hardware system, and the resulting observations were applied to refine the methodology and to formulate theories of N-version programming. A more detailed review of the UCLA research approach and a discussion of two sets of experimental results, one using 27 and the other using 16 independently written programs, have been published previously.[2,6]

The specification-oriented multiversion software experiment. Early research demonstrated the practicality of experimental investigation and confirmed the need for high-quality software specifications. The principal aim of the subsequent research was the investigation of software specification techniques. Other aims were investigating the types and causes of software design faults, proposing improvements both in software specification techniques, and in the use of those techniques, and proposing future experiments in the investigation of fault-tolerant design in software and in hardware.[11]

To examine the effect of specification techniques on multiversion software, an experiment was designed in which three different specifications were used.* The first was written in the formal specification language OBJ.[12] The second specification language used was the nonformal PDL, which was characteristic of current industry practice. The English language was used as the third, and "control," specification language, since English had been used in the previous studies.[6]

A specification is "formal" if it is written in a language with explicitly and precisely defined syntax and semantics.[13] Formal specifications have some very advantageous properties: they can be studied mathematically; they can be mechanized and tested to gather empirical evidence of their correctness; and they can be computer-processed to remove ambiguities, to eliminate inconsistencies, and to be made complete enough for empirical testing. The precision of formal languages makes writing rigorous specifications easier with them than with nonformal languages. OBJ was chosen for formal specification because the mechanism necessary to construct and test specifications with it was available at UCLA as were persons experienced in OBJ use. This personal experience proved to be important, since, as for all other formal specification languages examined, the existing documentation was quite inadequate. OBJ did, however, promote modularity and explicit handling of error conditions.

The nonformal specification language, PDL, lacks the power and sophistication of OBJ, but it does have adequate documentation, is reasonably well known and has been in use in industry for several years. Writing specifications in PDL is straightforward, while understanding them depends largely on the care taken by the writer. PDL provides extensive cross referencing and indexing—features that would be very useful in OBJ. Specifications written in PDL, however, do tend to be rather long.

The problem chosen for the experiment was an "airport scheduler" exercise. This database problem concerns the operation of an airport, in which flights are scheduled to depart for other airports and seats are reserved on those flights. The problem was discussed originally by Ehrig, Kreowski, and Weber[14] and later used to illustrate OBJ by Goguen and Tardo.[12] Because the problem is transaction-oriented, the natural choice of N-version cross-check points was at the end of each transaction. With the OBJ specification as a basis, one specification was written in PDL and another in English. The OBJ specification was 13 pages long, the English specification took 10 pages, and the PDL required 74 pages.

Execution of the experiment. Programmers with reasonable proficiency in PL/1 were recruited among the computer science students at UCLA. Each was assigned to work with one of the three specifications; no specification was tackled by a group whose overall range of abilities was not representative of the total range. The programmers were given a realistic deadline and a monetary incentive to produce programs of at least minimal quality by the deadline. The experiment proceeded in several steps:

(1) recruiting;
(2) teaching OBJ and PDL;
(3) examining and ranking;
(4) assigning the problem; and
(5) evaluating programs.

A seminar was held at UCLA to announce our need for participants, and we recruited 30 progammers whose abilities ranged from good to excellent, who were senior or graduate students, and whose professional experience ranged from none at all to several years. The next stage was the presentation of a one-day course on OBJ and PDL, which was necessary because the programmers lacked familiarity with OBJ and had very little familiarity with PDL. Study material was distributed and an examination was held two days later, at which the 30 participants were ranked as good, average, or poor. The members of each group were then assigned (in roughly equal numbers) to use the OBJ, PDL, and English specifications, respectively, in such a way as to avoid loading any of the specifications with either predominantly good or predominantly bad programmers.

*OBJ is a formal language for writing and testing algebraic program specifications developed at UCLA by Goguen from 1976 to 1979. It is also an applicative, nonprocedural programming language. PDL is a nonformal language for writing program specifications and designs. It was developed by Cain, Faber, and Gordon in the early 70s and functions as a production documentation tool.

At a subsequent meeting, each programmer was given a packet containing the specification; a notebook to record programmer effort, bugs encountered, and other problems; and a questionnaire on the specification and its use. It was also made clear that the programmers would not be paid for their work unless their programs passed a straightforward viability test. While an example of a typical viability test was given, the actual test to be used was not revealed. The programmers were requested not to discuss the project with other participants, and the goal of the experiment was once again carefully explained to support this request. Programmers were expected to analyze their specification and were allowed to ask questions individually. To preserve independence, however, group discussions were not held.

At the end of a four-week period, 18 of the 30 programmers returned working program versions of the airport scheduler written in PL/1. Of the 18 program versions, seven were written from the OBJ specification, five from the PDL specification, and six from English. All 18 programs were run with the standard viability test. After minor modifications were made to two programs by the original programmers, all 18 were judged satisfactory and were prepared for more detailed testing.

To conduct the more extensive testing, a very demanding set of 100 transactions was developed in an attempt to exercise as many features of the programs as possible. The immediate consequence of running the individual programs with these transactions was the discovery that 11 of the 18 programs aborted on invalid input. This had been noted as a very dangerous situation to encounter in *N*-version programming.[6] When executed in sets of three, one aborting version usually causes operating system intervention in all versions, effectively allowing the bad version to outweigh two otherwise healthy versions. To fix this situation, we provided all programs with an acceptance test that detected and attempted recovery from these otherwise catastrophic faults.

Results of the experiment

Program sizes and time requirements varied considerably (see Table 1). To conduct testing of individual versions, all 18 program versions were first executed with an 18-version decision algorithm in order to completely define the expected result for each of the 100 transactions. It would have been preferable to execute the OBJ specification to determine the expected result. Unfortunately, OBJ was not powerful enough to allow complete specification of all implementation constraints. This limitation is common to many formal specification languages.

Next, each version was executed alone, and the result of each case was compared to the expected result and classified as follows:

- "OK point," if the result was the expected result;
- "cosmetic error," if the result contained correct data but had, for example, misspelled text or contained bad formating;
- "good point," (code *G*) if the result was an OK point or a cosmetic error;
- "detected error" (code *D*), if the program version failed the acceptance test that had been added to detect and attempt recovery from abort conditions;
- "undetected error" (code *U* or *U**), if the version passed the acceptance test but was in error; in other words, errors were detectable only by comparison with the expected result. *U* is a distinct error that differs from the results of the other two versions, while *U** is an error that is similar for two or all three of the versions.

The individual test results are summarized in Table 2.

All 816 possible triplets of the 18 versions (see Table 3) were then executed with a three-version decision algorithm, using 100 transactions for each triplet. There were now three version results to consider for each transaction,

Table 1. Characteristics of the 18 versions of programs written in OBJ, PDL, and English by UCLA study participants. For each program version, the number of PL/1 statements used in the program (PL/1 STMTS), the number of procedures used (NO. PROCS), the compile time (COMPILE TIME), the execution time for the 100 transaction test case (EXEC. TIME), and the program size in bytes (SIZE BYTES) are shown. The time unit is a machine-unit second, which is a measure of time, and other resources, such as I/O needs.

NAME OF VERSION	PL/1 STMTS	NO. PROCS	COMPILE TIME	EXEC. TIME	SIZE BYTES
OBJ1	423	22	15.14	3.89	37600
OBJ2	400	28	11.35	3.96	28048
OBJ3	398	17	7.42	4.33	30904
OBJ4	328	14	8.62	4.77	29920
OBJ5	455	14	14.79	3.10	32304
OBJ6	243	16	4.71	2.70	20960
OBJ7	336	23	8.30	4.92	34808
PDL1	455	27	16.96	3.16	24928
PDL2	501	33	19.58	19.68	29656
PDL3	242	19	4.31	4.09	27360
PDL4	437	39	16.31	2.84	30896
PDL5	217	11	4.26	4.30	26440
ENG1	260	21	4.75	3.33	27552
ENG2	372	19	12.41	3.89	37792
ENG3	385	30	8.12	2.41	20648
ENG4	689	25	28.23	2.94	26864
ENG5	481	15	8.76	2.42	24056
ENG6	387	12	19.23	3.99	24656

Table 2. Test results for individual versions of the 18 programs written in OBJ, PDL, and English.

VERSION	OK POINTS	COSMETIC ERRORS	GOOD OK OR COS.	DETECTED ERRORS	UNDET. ERRORS
OBJ1	73	0	73	2	25
OBJ2	71	18	89	8	3
OBJ3	67	11	78	4	18
OBJ4	69	3	72	8	20
OBJ5	67	12	79	0	21
OBJ6	46	0	46	0	54
OBJ7	52	17	69	7	24
PDL1	59	2	61	1	38
PDL2	54	2	56	32	12
PDL3	95	0	95	4	1
PDL4	45	28	73	0	27
PDL5	94	0	94	5	1
ENG1	74	12	86	0	14
ENG2	67	27	94	0	6
ENG3	97	1	98	0	2
ENG4	30	5	35	25	40
ENG5	55	6	61	0	39
ENG6	53	3	56	9	35

with the results coded as shown above. Results of the types G, U, and U^* all passed the acceptance test and can only be classified here because the expected result has been determined in advance.

Now let us consider the decision algorithm that was used to produce one decision result from the three separate version results. Of the 64 possible triple combinations of the version result codes (G,D,U,U^*), 27 contain a single U^* code. Since a single U^* error is meaningless, these combinations are discarded to leave 37 remaining combinations. These fall into 14 separate categories, as illustrated in Table 4. We review these 14 categories with respect to the number of versions used by the decision algorithm. The detected errors D were excluded by the decision algorithm. Table 5 summarizes the decision results of all 81,600 triple computations.

Triplex decisions. The two cases in which all three results are similar, $V_1 = V(G,G,G)$ and $V_{14} = V(U^*,U^*,U^*)$ are indistinguishable to the decision algorithm and are both given the highest confidence rating (see Table 5). Of course, V_1 produced the expected result and V_{14} did not. Also, we expect three similar errors (V_{14}) to be rare. Actually, V_1 occurred in 44.9 percent of the decisions and V_{14} in only 0.1 percent.

Table 3. A list of the three-version triplets where *O* designates an OBJ version, *P* designates a PDL version, and *E* designates an English version.

TRIPLET COMPOSITION	NUMBER OF TRIPLETS
OOO	35
PPP	10
EEE	20
OPE	210
OOP	105
OOE	126
OPP	70
OEE	105
PPE	60
PEE	75
ALL	816

In three cases, two of the three results were similar and therefore outweighed the third dissenting version. The case $V_3 = V(G,G,U)$ produced the expected result and appeared in 27.1 percent of the decision results; $V_7 = V(G,U^*,U^*)$ allowed two similar errors to outweigh a good result and occurred 2.4 percent of the time; $V_{13} = V(U,U^*,U^*)$ occurred 0.4 percent of the time. Again these three cases were indistinguishable to the decision algorithm, and all were given a confidence level of "two."

In two cases, $V_6 = V(G,U,U)$ and $V_{12} = V(U,U,U)$, the decision algorithm identified three distinct version results and so declared that a decision result could not be obtained. V_6 occurred in 8.4 percent of the decisions and V_{12} in 1.7 percent. The confidence level was declared to be "zero."

Duplex decisions. When the acceptance test of a program version detected an internal error, it signaled the decision algorithm and its result was not used in the decision. The remaining two version results were used in a duplex decision.

In two cases, $V_2 = V(G,G,D)$ and $V_{11} = V(U^*,U^*,D)$, the results were similar and gave either the expected result (V_2) or an error (V_{11}). V_2 occurred in 6.5 percent of the decisions and V_{11} in 0.1 percent. Both cases were given a confidence level of "two."

In $V_5 = V(G,D,U)$ and $V_{10} = V(D,U,U)$, two distinct results remained after error detection and the decision algorithm declared that there was no decision result. These cases were said to have a confidence level of "zero" and occurred in 4.9 percent and 1.1 percent of the decisions, respectively.

Simplex decisions. When two versions failed their acceptance tests, the decision algorithm made the simplex decision of accepting the remaining result as valid with a confidence level of "one." Case $V_4 = V(G,D,D)$ gave the expected result and occurred in 1.6 percent of the decision results, while $V_9 = V(U,D,D)$ gave an undetected error and occurred in 0.6 percent of the decisions. The con-

fidence level of "one," which they were given, may be insufficient in some applications; the result in such instances would be declared not sufficiently dependable.

Null decisions. When all three versions failed acceptance tests, there was no decision result; consequently, the confidence level was declared "zero." Case $V_8 = V(D,D,D)$, the null decision case, occurred in 0.2 percent of the decisions.

Types and causes of errors. Several program versions produced many *cosmetic* errors, such as a result misalignment or a format misspelling, and a preprocessor to the decision algorithm was implemented to attempt to fix them. The cosmetic errors were caused by disregard of the specification and by *personalization* of the result. Personalization seems to occur in response to the programmer's need to leave a mark of individuality. Explaining that program results were to be computer processed and

Table 4. The three-version decision algorithm, where G = Good Point, D = Detected Error, and U or U* = Undetected Error (either distinct or similar, respectively). The confidence level describes how many version results the decision algorithm used in obtaining the decision result.

CASE	VERSION RESULTS	NO. OF ERRORS	DECISION ALGORITHM	DECISION RESULT	CONF. LEVEL
V1	G.G.G	0	TRIPLEX	G	3
V2	G.G.D	1	DUPLEX	G	2
V3	G.G.U	1	TRIPLEX	G	2
V4	G.D.D	2	SIMPLEX	G	1
V5	G.D.U	2	DUPLEX	D	0
V6	G.U.U	2	TRIPLEX	D	0
V7	G.U*.U*	2	TRIPLEX	U*	2
V8	D.D.D	3	NULL	D	0
V9	D.D.U	3	SIMPLEX	U	1
V10	D.U.U	3	DUPLEX	D	0
V11	D.U*.U*	3	DUPLEX	U*	2
V12	U.U.U	3	TRIPLEX	D	0
V13	U.U*.U*	3	TRIPLEX	U*	2
V14	U*.U*.U*	3	TRIPLEX	U*	3

Table 5. The results of decision algorithms.

		THREE-VERSION TRIPLET COMPOSITION				
CASE	ALL	000	PPP	EEE	OPE	OTHER
V1	36665	1703	448	820	9354	24340
	44.9%	48.7%	44.8%	41.0%	44.5%	45.0%
V2	5292	129	128	166	1341	3528
	6.5%	3.7%	12.8%	8.3%	6.4%	6.5%
V3	22105	842	274	554	5939	14496
	27.1%	24.1%	27.4%	27.7%	28.3%	26.8%
V4	1283	48	8	32	347	848
	1.6%	1.4%	0.8%	1.6%	1.7%	1.6%
V5	3986	141	88	80	1046	2631
	4.9%	4.0%	8.8%	4.0%	5.0%	4.9%
V6	6838	264	18	206	1747	4603
	8.4%	7.5%	1.8%	10.3%	8.3%	8.5%
V7	1944	86	12	82	386	1378
	2.4%	2.5%	1.2%	4.1%	1.8%	2.5%
V8	176	4	0	0	65	107
	0.2%	0.1%	0.0%	0.0%	0.3%	0.2%
V9	477	20	7	4	123	323
	0.6%	0.6%	0.7%	0.2%	0.6%	0.6%
V10	867	11	6	22	274	554
	1.1%	0.3%	0.6%	1.1%	1.3%	1.0%
V11	87	6	0	0	28	53
	0.1%	0.2%	0.0%	0.0%	0.1%	0.1%
V12	1415	173	1	20	275	946
	1.7%	4.9%	0.1%	1.0%	1.3%	1.7%
V13	353	48	0	8	62	235
	0.4%	1.4%	0.0%	0.4%	0.3%	0.4%
V14	112	25	10	6	13	58
	0.1%	0.7%	1.0%	0.3%	0.1%	0.1%
Total	81600	3500	1000	2000	21000	54100
	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%

thus had to be generated exactly as specified might have helped to reduce this problem.

All program versions contained acceptance tests that performed limited self-checking and, on error detection, sent an error message to the decision algorithm. These *detected* errors were not considered by the decision algorithm. They were caused mainly by inadequate input checks and other interpretation faults.

The *undetected* errors were of greatest interest. These errors passed a version's acceptance test and reached the decision algorithm. Undetected similar errors are particularly dangerous in a multiversion software system because they can outweigh a good version. Every undetected similar error in the experiment was traced back to its cause, which was found to be either (1) a specification fault, (2) an interpretation fault, or (3) an implementation fault.

Specification faults. A fault that occurs in the specification phase of software development is potentially very costly, since it often remains undetected until after completion of the software. Five specification faults were found and are shown in Table 6. One fault, *S*1, was traced to the OBJ specification and was actually more of a language inadequacy than a true specification defect. This fault was found in four of the seven OBJ programs. The other four faults were due to ambiguities in the English specification, and there were no faults in the PDL specification.

Interpretation faults. An interpretation fault is caused by a misinterpretation or misunderstanding of the specification, rather than by a mistake in the specification. We distinguish between interpretation faults caused by inadequate understanding of the specification—and implementation faults caused by faulty implementation of the specification. Failing to check input values, for example, is an interpretation fault, while being unable to retrieve records from the database is an implementation fault. There were nine interpretation faults (see Table 7), the most serious of which was the failure of all five PDL versions and one OBJ version to check input parameters adequately. Many of the interpretation faults could have been prevented by giving more examples in the specifications. Indeed, some programmers were unable to understand the specifications and relied, instead, on the examples given and on their intuition.

Implementation faults. An implementation fault is caused by a deficient implementation of the specification. There were seven implementation faults (see Table 8), of which four concerned the inability to correctly delete records in the database. Most of the implementation faults would have been detected by the programmers if more extensive testing had been performed. This was not encouraged, however, since one of our primary aims was to study the increase in reliability to be gained with imperfect programs used in *N*-version systems.

The causes of similar errors. In many cases, similar errors were caused by related faults. As an example, consider the transaction:

SCHEDULE 3, JFK, 1:00, 2, 727 (transaction 8)

This completely valid transaction was rejected by ENG4 and ENG6 because the departure time had not been input as "01:00" (fault *S*2).

A quite surprising result was that independent faults often produced similar errors. For example, consider transactions 3, 26, 36, and 69:

LIST (transaction 3)

After processing this, OBJ6 did no further processing of transactions (fault *I*8).

SCHEDULE 10, SAC, 23:00, 10, DC10 (transaction 26)

Because the destination code was not one of four sample codes, programs OBJ1, OBJ4, OBJ5, and OBJ7 would not accept his transaction (fault *S*1).

CANCEL 999 (transaction 36)

Because of a faulty cancel algorithm, this transaction was not performed correctly by OBJ3 (fault *L*2). Next, consider the following transaction

SCHEDULE 6, LAX, 20:30, 10, 727 (transaction 69)

This transaction should have been rejected, because it used an illegal duplicate plane number, but all the OBJ versions listed above (OBJ1, OBJ3, OBJ4, OBJ5, OBJ6, OBJ7) accepted this transaction because of earlier errors made in transactions 3, 26, or 36. Here, three independent design faults are illustrated: the arbitrary decision of one programmer to shut down after a certain sequence of transactions had been processed (fault *I*8); a poor Cancel algorithm (fault *L*2); and the acceptance of only four destinations (fault *S*1). These all led to similar errors, in which an illegal transaction was accepted and simply acknowledged. This particular result, simple acknowledgment, contained no data and was the default result of several different transactions. A partial solution to this problem is never to allow a simple acknowledgment, since the acknowledgment may cause a similar error. Additional supporting data such as identification of partial program state should always be included and should lead to fewer similar errors.

Need for improvement in specification techniques

Despite real strengths, formal software specification techniques often have not paid off for conventional designs[15] and produce specifications of limited value. Some areas of concern and suggested improvements are presented here from our experiences with OBJ.

Lack of experience and documentation. Formal specification languages are difficult to use, hard to write, and often sensitive to typographical errors.[12] Programmers are usually unfamiliar with the applicative style of most formal specifications; documentation is inadequate; and training material does not exist. These problems are to be expected during the early stages of development of any new methodology and are probably transitory.

Automated tools. Current specification techniques, while capable of being computer processed, provide little support and few tools. A cross-referencing facility is par-

ticularly needed to keep track of function definitions and subsequent invocations. Databases containing predefined common objects and lower level specifications would be helpful, as would the capability of viewing the specification at different levels of abstraction.

Applicability to the problem domain. In order to produce a straightforward specification, the language must be appropriate for the particular problem domain. Existing specification languages are limited and cumbersome, partly because of the disparity between them and the problem and implementation domains. It is very difficult, for example, to specify implementation constraints and interface specifications. Since a specification should describe what is to be done, it is important to address known implementation constraints as early as possible. The new Larch family of specification languages[16] is a step in the right direction.

Limited power. The lack of built-in functions puts a large burden on the writer and, by cluttering the specification with low-level detail, makes it more difficult to comprehend. Support for the specification of real arithmetic has been missing from formal specification languages, making numerical problems very difficult to define.

Specifying temporal characteristics. Specification languages have not allowed us to define notions of time and concurrency. Specifying the nondeterminacy for asynchronous systems remains a difficult research problem.[13,17] It is particularly important to be able to specify timing in a multiversion software environment; timing specifications are needed to address performance requirements for individual versions, as well as for the specification of the decision algorithm and its associated watchdog timers that prevent a failed version from hanging up the system.

Language syntax. OBJ programmers were unanimous in their dislike of the language's syntax, even though the specification that they received was a substantial improvement over a "raw" OBJ specification. The OBJ interpreter treats a specification as a continuous, Lisp-like character string, making parsing very difficult. The specification used had been post-processed by a text editor to make it a little more intelligible.

Abstraction versus procedural specifications. More experience is needed in writing and using true abstract specifications as well as "rapid prototypes." What are the consequences of introducing bias in a "rapid prototype," and are the more difficult abstract specifications worth the extra effort? This experiment showed that, at the current level of development, a familiar procedural specification, PDL, was easier to use and understand than the more difficult, unfamiliar, abstract specification of OBJ.

Our study of both formal and informal specification languages has shown that none of the languages available is sufficiently automated. Formal specification languages, while showing promise for the future, thus far have been very difficult to use and understand and are severely limited in power.

Work in progress and goals for long-range research

One major goal of the first-generation experiments just described was to apply our accumulated experience to the next generation of experiments. It has become evident that the general UCLA campus computing facility is a nonsupportive environment for multiversion software experiments. To establish a long-term research facility for further experimental investigations in the design and evaluation of N-fold-implemented fault-tolerant systems, we are designing a multichannel system testbed at the UCLA Center for Experimental Computer Science. The testbed will be a part of the center's advanced local network, which utilizes the Locus distributed operating system[18] to operate a set of 20 Vax 11/750 computers.

A distributed system testbed for multiversion software. The testbed is intended to provide two distinct research and utilization opportunities. First, it will serve as an experimental vehicle for future fault-tolerant design investigations; and second, it will be a very high-reliability

Table 6. Specification faults causing similar errors.

SPEC. FAULT	APPEARS IN OBJ	PDL	ENG	DESCRIPTION
S1	1.4.5.7			Only defines four destinations
S2			4.5.6	Time shown as 09.45 in example
S3			4.5.6	Error message order ambiguity
S4			2.4	Duplicate error message ambiguity
S5			6	Parameter checking ambiguity

Table 7. Interpretation faults causing similar errors.

INT. FAULT	APPEARS IN OBJ	PDL	ENG	DESCRIPTION
I1	2	1.2.3.4.5		Did not check input parameters first
I2		1.4		Expects time as 09:45
I3	1.2.5	4	1	Wrong error message on illegal input
I4	4	4	1	Create does not work the second time
I5		2.4	5	Error message output on legal input
I6	3	1	6	Allows null parameters
I7		1	4	Allows invalid parameters
I8	6			No output after first list produced
I9		2		Cannot handle invalid input

Table 8. Implementation faults causing similar errors.

IMPL. FAULT	APPEARS IN OBJ	PDL	ENG	DESCRIPTION
L1	1			Cancel does not work on last entry
L2	3		5	Cancel works only on partial database
L3	7			Cancel unknown cancels last entry
L4		1		Cannot retrieve record
L5		1		Allows duplicate record
L6			4	Cancel, reserve-seat ignored
L7			4	Bad input leads to chaos

and continuous availability node for other users of the local network who have a need for exceptionally reliable computing.

Certain requirements for designing a distributed system presented themselves to us. It is necessary to provide independent simultaneous execution and to compare intermediate results when the programming versions have reached identical cross-check points. A synchronization mechanism is needed that makes sure that the results compared are from identical *cc*-points. Because of reliability requirements, the synchronization must operate without any centralized components. To obtain such a mechanism in a distributed system is difficult. The Locus distributed operating system communicates by passing messages with inherently varying communication delays that are significant when compared with the elapsed time between events in a single process. As a consequence of these characteristics, a consistent view of the global state does not exist at all times.

For these reasons, the testbed was designed as a set of hierarchical layers, as shown in Figure 1. A layered approach has several advantages. It reduces complexity, shields higher layers from details of how the services offered by lower layers are actually implemented, and facilitates verification of the system. Each layer uses the services provided by the lower layers and adds new services, which are used in turn by the higher layers. The testbed is composed of the following four layers:

(1) *The transport layer,* which moves messages from one site to another or within one site. Its goal is to make sure that no message is lost, duplicated, damaged, or misaddressed. It is designed so that messages arrive in the order that they are sent. The transport layer can also set up connections between authorized entities, such as the transport layers, at the sites.

(2) *The synchronization layer,* which collects messages (that hold version results) from identical cc-points, delivers them to the transport layer, and synchronizes the restart of the versions. This layer can establish communication connections between versions.

(3) *The decision and local-executive layer.* This layer processes version results from identical cc-points by using the decision algorithm and determines whether a version is faulty or not. The decision result and the version fault status are forwarded to the synchronization layer to be used in further computations.

(4) *The user and global-executive layer.* This layer consists of two parts: the multiple versions of user programs and a replicated global executive program that has the same functions as the global executive found in SIFT.[5] It performs the fault-tolerance functions that are not performed synchronously at each cc-point. These functions include system fault diagnosis, reconfiguration, and fault reporting for maintenance purposes.

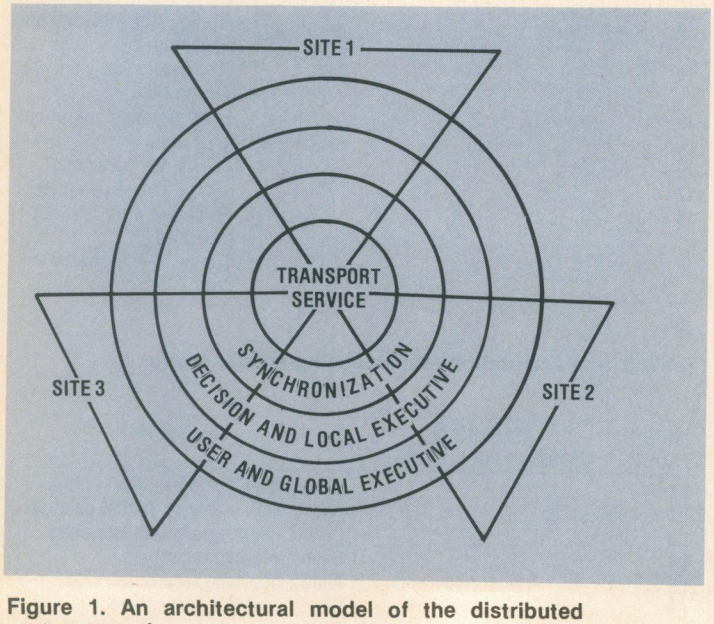


Figure 1. An architectural model of the distributed system tested.

A functional diagram of the current system is given in Figure 2. *Sync* represents the synchronization module; *SV*, the supervisory module that performs local executive functions; *DEC,* the decision algorithm; and *Vn* represents version *n* of the user program. The global executive is not shown. The *pipe* and *process* systems of Locus are used to connect sites and to provide reliable communication. The usual operating system functions (file system, process management, and conventional interprocess communication) of Locus are available to the user software. The testbed will therefore support the operation of realistically large and complex programs. An initial implementation of the synchronization, supervisory, and decision algorithm modules has been written in the C programming language.

Enhanced fault-tolerance mechanisms. As the testbed evolves, we plan to investigate various fault-tolerance mechanisms. The present implementation of the decision algorithm will be replaced by a more general decision algorithm. Two candidate algorithms are *skew* and *adaptive voting.*[6] Skew voting handles numerical results that are unequal because of the differences in precision allowed around some unknown reference value. Adaptive voting weights version results; the weights can then be dynamically calculated to favor similar results. The decision algorithm will also have some ability to recognize cosmetic errors and deal with diverse data representations.

The user interface is another important and difficult problem that will be addressed. Functions that must be supplied include distribution and execution of program versions, log recording for each trial, statistics generation, and interactive version and system manipulation. By providing and evaluating these, we expect to obtain quantitative experimental decision 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 will be thoroughly examined is the recovery of failed versions through rollback, rollforward, 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.

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 Locus implementation of the testbed 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. In addition, Locus can be used to simulate real-time execution.

Use of advanced specification languages. Significant progress has occurred in the development of formal specification languages since our previous experiments. The promising languages Larch,[16] Clear,[19] and Prolog are being evaluated for application in future experiments. We have been impressed by the ease with which Prolog specifications can be written even without such features as abstract data types and implicit-control structure.

Once the testbed is operational, we will have an initial statement of the attributes required for the components of the working fault-tolerance mechanisms implemented in C. The next step will be a formal specification of the protocol layers, the decision algorithm, and the local and global executives. We plan to investigate the use of Concurrent Prolog as the first specification language. The specification will provide an executable rapid prototype of the underlying supervisory operating system and the application versions. It will allow not only the migration to real-time systems, but also the use of multiversion software techniques for the fault-tolerance mechanisms themselves. The goal is an operating system that supports design diversity in application programs and which is itself diverse in design.

Independent specifications of some operating system modules in Larch and Clear will serve to compare their merits with those of Prolog. Further research is planned in the application of dual-diverse formal specifications to eliminate errors traceable to specification faults and increase reliability.

Software cost studies. It has been estimated that 70 percent of the overall cost of software is spent on software maintenance, which includes (1) correction of bugs and (2) enhancement due to changing requirements. We will carefully monitor the cost of producing program versions during all phases of development.

The cost of removing bugs. By combining software versions that have not been subjected to V&V testing to produce highly reliable multiversion 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 production use of the system. The software faults then can be fixed without affecting system availability.

The cost of enhancement. 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 insight into the criteria to be used for a choice.

Mail-order software experiments. To test 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 by having them 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. A model experiment has been specified, which includes 15 general guidelines and 10 specific tasks.[20]

The first-generation experiments described here have provided a proof-of-concept and have shown the feasibility of design diversity as a complement to fault avoidance. Whether it is a practical and cost-effective general solution to the design fault problem remains to be seen. However, tolerance of design faults in software and in hardware is the challenge of the eighties, and the results presented here encourage us to propose further experimentation. To this end, second-generation experimentation is now underway at four universities* to measure the efficacy of design diversity and to demonstrate conclusive reliability increases under large-scale, controlled experimental conditions. *

Acknowledgments

This research has been supported by NSF grants MCS-78-18918 and MCS 81-21696, and by a research

*Experimentation is now underway at UCLA, the University of Virginia, the University of Illinois, and North Carolina State University.

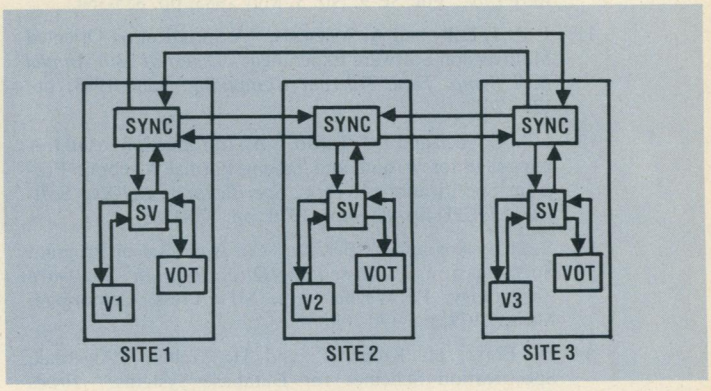


Figure 2. A functional block diagram of the present system in which SYNC = Synchronization module, SV = Supervisory module, Vn = Version *n* of user's application program, VOT = Voting module, and right and left arrows indicate communication lines (provided by the "pipe" system call).

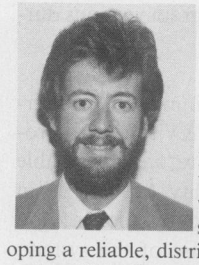
grant from the Battelle Memorial Institute. Current support is provided by the Advanced Computer Science program of the Federal Aviation Administration. Valuable contributions to the research have been made by Liming Chen, Baron Grey, Per Gunningberg, Srinivas Makam, Tetsuichiro Sasada, Lorenzo Strigini, and Pascal Traverse. This article has benefited from perceptive comments by Flaviu Cristian and Jack Goldberg. *

References

1. A. Avižienis, ed., "Special Issue on Fault-Tolerant Digital Systems," *Proc. IEEE*, Vol. 66, No. 10, Oct. 1978.
2. A. Avižienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," *Proc. Compsac 77*, Nov. 1977, pp. 149-155.
3. T. Anderson and P.A. Lee, *Fault Tolerance, Principles, and Practice*, Prentice-Hill International, London, England 1981, pp. 249-291.
4. F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Trans. Computers*, Vol. C-31, No. 6, June 1982, pp. 531-539.
5. J. Goldberg, "SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control," *Proc. IFIP Congress 80*, pp. 151-156.
6. L. Chen and A Avižienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Digest of Eighth Annual Int'l Conf. Fault-Tolerant Computing*, June 1978, pp. 3-9
7. P. Morrison and E. Morrison, eds., *Charles Babbage and His Calculating Engines*, Dover Publications, Inc., New York, 1961, p. 177.
8. L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection Systems: An Experiment," *Proc. IFAC Workshop Safecomp 79.*
9. W. C. Carter, "Architectural Considerations for Detecting Run Time Errors in Programs," *Digest of 13th Annual Int'l Symp. Fault-Tolerant Computing*, June 1983, pp. 249-256.
10. T. Anderson and J. C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Trans. Software Eng.*, Vol. SE-9, No. 3, May 1983, pp. 355-364.
11. J. P. J. Kelly and A. Avižienis, "A Specification-Oriented Multiversion Software Experiment" *Digest of 13th Annual Int'l Symp. Fault-Tolerant Computing*, June 1983, pp. 120-126.
12. J. A. Goguen and J. J. Tardo, " An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," *Proc. Specifications Reliable Software,* 79CH1401-9C, Apr. 1979, pp. 170-189.
13. B. H. Liskov and V. Berzins, "An Appraisal of Program Specifications," *Research Directions in Software Technology*, P. Wegner, ed., MIT Press, Cambridge, Mass., 1979, pp. 170-189.
14. H. Ehrig, H. Kreowski, and H. Weber, "Algebraic Specification Schemes for Database Systems," *Proc. Fourth Int'l. Conf. Very Large Databases*, Sept. 1978, pp. 427-440.
15. D. L. Parnas, "The Role of Program Specification," *Research Directions in Software Technology*, P. Wegner, ed., MIT Press, Cambridge, Mass., 1979, pp. 364-370.
16. J. V. Guttag and J. J. Horning, "An Introduction to the Larch Shared Language," *Proc. IFIP Congress 83*, pp. 809-814.
17. P. M. Melliar-Smith, "System Specifications," *Computing Systems Reliability*, T. Anderson and B. Randell, eds., Cambridge University Press, Cambridge, England, 1979.
18. G. Popek et al., "LOCUS—A Network Transparent, High Reliability Distributed System," *UCLA Computer Science Department Quarterly*, Vol. 9, No. 1, Winter 1981, pp. 75-88.
19. R. M. Burstall and J. A. Goguen, "An Informal Introduction to Specifications Using CLEAR," *The Correctness Problem in Computer Science*, R. Boyer and H. Moore, eds., Academic Press, New York, 1981, pp. 185-213.
20. J. P. J. Kelly, "Specification of Fault-Tolerant Multiversion Software: Experimental Studies of a Design Diversity Approach," *UCLA Computer Science Department,* tech. report CSD-820927, Sept. 1982.

Algirdas Avižienis is currently the chairman of the Computer Science department of UCLA, where he has served on the faculty since 1962. He also directs the UCLA Reliable Computing and Fault-Tolerance research group, which he established at UCLA in 1972. Current projects of the group include design diversity for the tolerance of design faults, fault tolerance in distributed systems, and fault-tolerant supercomputer architectures. Avižienis also directed research on fault-tolerant spacecraft computers at Caltech's Jet Propulsion Laboratory in Pasadena, California (1960 to 1973). This effort resulted in the experimental JPL Self Testing And Repairing, or STAR, computer (*IEEE Trans. on Computers,* Vol. C-20, No. 11, 1971). For his pioneering work in fault-tolerant computing Avižienis was elected Fellow of IEEE, received the NASA Apollo Achievement Award, the Honor Roll award of the IEEE Computer Society, the AIAA Information Systems Award, and the NASA Exceptional Service Medal.

Avižienis received his BS (1954), MS (1955) and PhD (1960) in electrical engineering from the University of Illinois. He served as founding chairman of the IEEE-CS Technical Committee on Fault-Tolerant Computing (1969-1973), and of IFIP Working Group 10.4, "Reliable Computing and Fault Tolerance" (1980-1981).



John P. J. Kelly is a visiting assistant professor of the Computer Science department of UCLA and a member of the UCLA Reliable Computing and Fault-Tolerance research group. His primary research interests include the use of design diversity to provide fault tolerance in software, fault-tolerance in distributed systems, and software engineering. Kelly is also a computer scientist at Ordain, Inc. where he is developing a reliable, distributed database machine.

Since 1972, Kelly has been a consulting computer scientist in the areas of fault tolerance and distributed systems to various organizations both in the USA and abroad, including the NASA-Langley Research Center, the Jet Propulsion Laboratory, Transaction Technology Incorporated, and Telos Computing Corporation.

Kelly received his BS (1975) and MS (1977) degrees in mathematics from the University of Cambridge, England and his PhD (1982) in computer science from UCLA.

Questions about this article can be directed to either author, 3731 Boelter Hall, University of California, Los Angeles, CA 90024.