Design diversity: An approach to fault tolerance of design faults

by ALGIRDAS AVIZIENIS
University of California, Los Angeles
Los Angeles, California

ABSTRACT

Diversity of design is discussed as a means to attain fault tolerance with respect to latent design faults in software and hardware. Some potential advantages of this approach in software versus a single design protected by fault avoidance (verification, validation, and proofs) are presented. An extension to design fault tolerance in VLSI circuits is identified. The results of earlier experimental studies are reviewed, and new results of a specification-oriented multiversion software experiment are summarized.
INTRODUCTION: THE DESIGN DIVERSITY APPROACH

Over the past two decades, several successful fault-tolerant systems (tolerating faults of physical origin, to be called “physical faults” in this paper) have been designed, built, and used in important applications. Major examples are the JPL-STAR (Self-Testing And Repairing) computer for multi-year interplanetary space missions, the Bell Laboratories duplexed ESS central processors, and the advanced SIFT and FTMP designs intended to serve as real-time control computers for commercial airliners of the future. The SIFT and FTMP designs use a minimum of three complete and separate computing channels with majority voting (by software in SIFT; by hardware in FTMP) to assure system survival after the first physical fault. Reconfiguration and sparing are then used to lower the probability of system failure to the desired value of $10^{-9}$ for a 10-hour flight.

In contrast to the successful systems that exercise tolerance of physical faults, there are no examples of operational systems that tolerate design faults either in software or in hardware. The fault-avoidance approach is exclusively used to eliminate design faults. The inevitable left-over design faults are removed by maintenance procedures applied off-line, i.e., after a system crash has occurred. The question whether design faults can be successfully tolerated by extensions and generalizations of fault tolerance techniques has remained unanswered. The question can be addressed in two parts:

1. Is it possible to implement design fault tolerance regardless of cost?
2. Can this approach compete, with respect to cost, with the currently prevalent design fault-avoidance approaches that use verification, validation, and correctness proofs?

In setting out to investigate the potential of fault tolerance techniques in the domain of design faults, we note that a strong analogy exists between physical and design faults, as shown in Figure 1.

The existence of systems with strong tolerance of physical faults attained through multiple-channel computing is an encouraging fact. However, it is evident that the channels are identical and therefore do not possess the critically important property of design diversity that is needed to tolerate the manifestation of a latent design defect. Clearly, the multiple computing channels will have the potential for design fault tolerance only if there is a very high probability that the left-over design faults do not evoke the same forms of undesirable behavior in a majority of channels; that is, if their symptoms are not isomorphic at the points of observation.

Consequently, design diversity is the new key requirement for design fault tolerance that needs to be added to a multi-channel system that tolerates physical faults. Design diversity in this context means the independent generation of two or more software or hardware elements (e.g., program modules, VLSI circuit masks, etc.) to satisfy a given requirement. It must be noted that the discussion of diversity applies not only to the initial generation of programs and designs but also to subsequent modifications or redesigns that are made in order to improve performance or to correct discovered defects and inadequacies.

CONDITIONS FOR THE INDEPENDENCE OF DESIGN FAULTS

Independence of the design and implementation efforts is the mechanism that is employed to minimize the probability of identical design-fault symptoms in a majority of computing channels. It is approached first by the use of different algorithms, programming languages, translators, design automation tools, implementation techniques, machine languages, and so on. The second condition for independence is the employment of independent programmers or designers, preferably with diversity in their training and experience.

The third and most critical condition for independence of design faults is the existence of a complete and correct initial statement of the requirements to be met by the diverse designs. This is the hard core of the fault-tolerance approach. Latent defects, such as inconsistencies, ambiguities, and omissions in the initial statement, are likely to bias otherwise en-

![Figure 1—An analogy between physical and design faults](From the collection of the Computer History Museum (www.computerhistory.org))
tirely independent programming or logic design efforts so that they produce isomorphic design faults.

The most promising approach to create the initial statement is the use of formal, very-high-level specifications that themselves can be automatically tested for latent defects, or even proven to be defect-free. Here perfection is required only at the highest level of specification; the rest of the design and implementation process and its tools are not required to be perfect, but only as good as possible within existing constraints on resources and time.

POTENTIAL ADVANTAGES OF DESIGN DIVERSITY

The most immediate and direct application of design fault tolerance through design diversity exists in the multichannel systems with very complete tolerance of physical faults (e.g., SIFTF23) that are employed in life-critical applications. The hardware resources and architectural features to support design diversity are already present, and implementation of design diversity is a logical extension of the existing physical fault tolerance mechanisms. Furthermore, design faults in the hardware of a channel can be tolerated by choosing for each channel functionally compatible hardware building blocks from different suppliers.

A more speculative, but also much more general application of design diversity is its use as a partial replacement for current software verification and validation (V&V) procedures. Instead of a thorough V&V of a single program, two independent versions are to be executed in an operational environment, completing V&V concurrently with productive operation. The doubled cost of producing the software is compensated by a reduction of the V&V time and a decrease in the cost of manpower and special tools needed for the very thorough V&V effort. The second (backup) version can be taken off line when adequate reliability of operation is reached, and then returned for special operating conditions that require greater reliability assurance, especially after modifications or after maintenance. A potential system lifetime cost reduction exists because such a system can support continued operation after latent design faults are uncovered, providing near 100% availability. The cost of fault analysis and elimination should be reduced because of the lesser urgency of the repair actions, since operation is not interrupted as long as the majority of channels are not affected.

A very intriguing long-range implication of the design diversity approach in software is the possibility of using a “mail-order” approach to the production of two or more versions of software modules. Given a precise formal specification that includes a set of fundamental tests, the software can be generated by programmers working at their own preferred times and locations, possibly using their own personal computing equipment. Two potential advantages have been identified:

1. The overhead cost of programming that accrues in highly controlled professional programming environments would be drastically reduced through this approach, which allows free play to individual initiative and uses low-cost home facilities.

2. The potential of the rapidly growing number of computer hobbyists to serve as productive programmers would be tapped through this approach. For various reasons, many individuals with programming talents cannot fill the role of a professional programmer as defined by today’s rigorous approaches to quality control and use of centralized sites during the programming process.

Finally, an important reliability and cost advantage through design diversity may be expected for VLSI circuit design. The growing complexity of VLSI circuits, with 400,000 gates/chip available today and 1 million gates/chip predicted for 1986, raises the probability of latent design faults, since a complete verification of the design becomes very difficult to attain. Furthermore, the design automation and verification tools themselves are subject to latent design faults. Even with multichannel fault-tolerant system designs, a single latent design fault would require the replacement of all chips of the class, since on-chip modifications are impractical. Such a replacement would be a costly and time-consuming process. On the contrary, use of design diversity of VLSI circuits does allow the continued use of chips with design faults, as long as their symptoms are not isomorphic at the circuit boundaries. Reliable operation throughout the lifetime of a system may be obtained by means of design diversity without having a single chip with a perfect design and without any modification of the basic structure of the VLSI circuits.

INITIAL STUDIES OF MULTIVERSION SOFTWARE: AN EXPERIMENTAL APPROACH

The potential advantages that were identified in the preceding section have provided the motivation for study of design diversity and design fault tolerance as alternatives to the generally used verification, validation, and proof methodology that aims to deliver perfect software products and hardware circuits.

An increasing awareness of the need for design fault tolerance led to the initiation of a research effort at UCLA in 1975. The work was founded on a 14-year background of continuous investigations in tolerance of physical faults, and its goal was to study the feasibility of adapting to software design fault tolerance the technique of N-fold modular redundancy (NMR) with majority voting, which is effective in the tolerance of physical faults. The approach was called N-version programming (NVP), and the first experimental study of its feasibility was completed in 1978. A literature search in 1975 revealed few other efforts in this area. Suggestions that this approach might be a viable method of software fault tolerance had been published recently. However, quite arguably, the first suggestion on record has been made by Dr. Dionysius Lardner, who wrote in his article “Babbage’s Calculating Engine,” published in the Edinburgh Review, No. CXX, July 1834, as follows:

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. 26

A second approach already under investigation in 1975 was the recovery block (RB) technique, in which alternate software routines are organized in a manner similar to the dynamic-redundancy (standby-sparing) technique in hardware. 30 The prime objective is to perform run-time software design fault detection by an acceptance test and to implement recovery by taking an alternate path of execution. This technique is also being continuously investigated at several locations. Some comparisons of RB with NVP have been made. 11,16 Several related research activities have been reported more recently. 18,24,27,34

N-version programming is defined as the independent generation of N ≥ 2 software modules, called versions, from the same initial specification. 2 Independent generation here means that programming efforts are carried out by individuals or groups that do not interact in the programming process. Wherever possible, different algorithms and programming languages or 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. The initial specification also states all the special features that are needed in order to execute the set of N versions in a fault-tolerant manner. 11 An initial specification defines (1) the function to be implemented by an N-version software unit; (2) data formats for the special mechanisms (c-vectors), comparison status indicators (cs-indicators), and synchronization mechanisms; (3) the cross-check points (cc-points) for c-vector generation; (4) the comparison (matching or voting) algorithm; and (5) the response to the possible outcomes of matching or voting. We note that comparison is used as a general term, while matching refers to the N = 2 case, and voting to a majority decision with N > 2. The comparison algorithm explicitly states the allowable range of discrepancy in numerical results, if such a range exists.

It is a fundamental conjecture of the N-version approach that the independence of programming efforts will greatly reduce the probability of identical software design faults occurring in two or more versions. Together with a reasonable choice of c-vectors and cc-points, this is expected to turn N-version programming into an effective method to achieve tolerance of software design faults. The effectiveness of the entire approach depends on the validity of this conjecture, so an experimental investigation was deemed to be important since, like all other formal specification languages examined, it had quite inadequate existing documentation.

The following software specification languages were examined as candidates for use in the experiment: OBJ, 19 SPE­CIAL, 32 DREAM, 31 SEMANOL, 1 UDSS, 9 and PDL. 10 Key attributes required for selection were comprehensibility, testability, maintainability, explicit handling of error conditions, and availability for immediate use.

To examine the effect of specification techniques on multiversion software an experiment was designed in which three different specifications were used. The first was the formal specification language OBJ. 19 The second specification language was used was the nonformal PDL that was characteristic of current industry practice. English language was used as the third, or control, specification language, since English had been used in the previous studies. 11

A specification is formal if it is written in a language with explicitly and precisely defined syntax and semantics. 26 This leads to some very advantageous properties: the specification can be studied mathematically; it can be mechanized and tested to gather empirical evidence of its correctness; it can be computer processed to remove ambiguities, to remove inconsistencies, and to be made complete enough (at least) for empirical testing; the interpretation by implementors and customers in an unambiguous way is easier; and writing rigorous specifications is easier with a formal methodology. OBJ was chosen as the formal specification language because the mechanism necessary to construct and test specifications using OBJ was available at UCLA along with local expertise. This proved to be important since, like all other formal specification languages examined, it had quite inadequate existing documentation. 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, the ease of understanding depending largely
on the amount of care taken by the writer. PDL provides
extensive cross referencing and indexing—a feature that
would be very useful in OBJ. Specifications written in PDL do
tend to be rather long, however.

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
Weber13 and later used to illustrate OBJ by Goguen and
Tardo.16 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 reference
point, a specification was written in PDL and another one in
English.

EXECUTION OF THE EXPERIMENT

Programmers with reasonable proficiency in PL/1 were re-
cruited among the Computer Science students at UCLA.
They were assigned to work with one of the three specifica-
tions; 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 mon-
etary incentive to produce programs of at least minimal qual-
ity by the deadline. The experiment proceeded in several
steps: (1) recruiting, (2) teaching OBJ and PDL, (3) exam-
ining and ranking, (4) Assigning the problem, and (5) evaluat-
ing programs.

A seminar was held at the UCLA Computer Science De-
partment to announce the need for programmers; and over
the next four weeks 30 programmers were recruited, whose
abilities ranged from good to excellent, who were senior or
graduate students, and who had anywhere from no profes-
sional experience at all up to several years of experience. The
next stage was the presentation of a one-day course on OBJ
and PDL, which was necessary because of the total lack of
familiarity with OBJ and 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 ranking were
then assigned in roughly equal numbers to use the OBJ, PDL,
and English specifications. The purpose of the examination
was to avoid loading any of the specifications with either
predominantly good or predominantly bad programmers.

At a subsequent meeting each programmer was given a
packet containing the specification, a notebook to record pro-
gramer 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 accept-
ance test. While an example of a typical acceptance test was
given, the actual test to be used was not revealed. They were
strongly requested to avoid working with other participants,
and the goal of the experiment was once again carefully ex-
plained to support this request.

At the end of the four-week interval 18 of the 30 pro-
grammers 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 acceptance test data. After minor modifica-
tions were made to two programs by the original program-
ners, all 18 were judged satisfactory and were prepared for
more detailed testing.

To conduct the more extensive testing, a very demanding
set of 100 input transactions was developed in an attempt to
exercise as many features of the programs as possible. The
immediate consequence of running the programs with this
input data was the discovery that 11 of the 18 programs
aborted on invalid input. This is, of course, a very dangerous
situation to encounter in N-version programming, as Chen
had found out.13 In this case, one aborting bad version usually
causes operating system intervention for all versions, effect-
ively allowing the bad version to outvote two otherwise
healthy versions. To fix this situation all programs were instru-
mented using PL/1 language capabilities to detect and to
attempt recovery from these otherwise catastrophic errors.
After such instrumentation, all programs survived the test
case input, with 10 of the 11 previously abortable programs
making reasonable recoveries.

Program size and time requirements varied considerably.
Table I shows, for each program version, the number of

<table>
<thead>
<tr>
<th>Version</th>
<th>PL/1 Stmts</th>
<th>Procs</th>
<th>PL/1 MUs</th>
<th>GO MUs</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJ1</td>
<td>423</td>
<td>22</td>
<td>15.14</td>
<td>3.89</td>
<td>37600</td>
</tr>
<tr>
<td>OBJ2</td>
<td>400</td>
<td>28</td>
<td>11.35</td>
<td>3.96</td>
<td>28048</td>
</tr>
<tr>
<td>OBJ3</td>
<td>398</td>
<td>17</td>
<td>7.42</td>
<td>4.33</td>
<td>30904</td>
</tr>
<tr>
<td>OBJ4</td>
<td>328</td>
<td>14</td>
<td>8.62</td>
<td>4.77</td>
<td>29920</td>
</tr>
<tr>
<td>OBJ5</td>
<td>455</td>
<td>14</td>
<td>14.79</td>
<td>3.10</td>
<td>32304</td>
</tr>
<tr>
<td>OBJ6</td>
<td>243</td>
<td>16</td>
<td>4.71</td>
<td>2.70</td>
<td>20960</td>
</tr>
<tr>
<td>OBJ7</td>
<td>336</td>
<td>23</td>
<td>8.30</td>
<td>4.92</td>
<td>34808</td>
</tr>
<tr>
<td>PDL1</td>
<td>455</td>
<td>27</td>
<td>16.96</td>
<td>3.16</td>
<td>24928</td>
</tr>
<tr>
<td>PDL2</td>
<td>501</td>
<td>33</td>
<td>19.58</td>
<td>19.58</td>
<td>29656</td>
</tr>
<tr>
<td>PDL3</td>
<td>242</td>
<td>19</td>
<td>4.31</td>
<td>4.09</td>
<td>27360</td>
</tr>
<tr>
<td>PDL4</td>
<td>437</td>
<td>39</td>
<td>16.31</td>
<td>2.84</td>
<td>30896</td>
</tr>
<tr>
<td>PDL5</td>
<td>217</td>
<td>11</td>
<td>4.26</td>
<td>4.30</td>
<td>26440</td>
</tr>
<tr>
<td>ENG1</td>
<td>260</td>
<td>21</td>
<td>4.75</td>
<td>3.33</td>
<td>27552</td>
</tr>
<tr>
<td>ENG2</td>
<td>372</td>
<td>19</td>
<td>12.41</td>
<td>3.89</td>
<td>27792</td>
</tr>
<tr>
<td>ENG3</td>
<td>385</td>
<td>30</td>
<td>8.12</td>
<td>2.41</td>
<td>20648</td>
</tr>
<tr>
<td>ENG4</td>
<td>689</td>
<td>25</td>
<td>28.23</td>
<td>2.94</td>
<td>26864</td>
</tr>
<tr>
<td>ENG5</td>
<td>481</td>
<td>15</td>
<td>8.76</td>
<td>2.42</td>
<td>24056</td>
</tr>
<tr>
<td>ENG6</td>
<td>387</td>
<td>12</td>
<td>19.23</td>
<td>3.99</td>
<td>24656</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
### TABLE II—Test results for individual versions

<table>
<thead>
<tr>
<th>Version</th>
<th>OK Points</th>
<th>Cosmetic Errors</th>
<th>Good OK+Cos</th>
<th>Detected Errors</th>
<th>Undet. Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORT1</td>
<td>73</td>
<td>0</td>
<td>73</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>ORT2</td>
<td>71</td>
<td>18</td>
<td>89</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>ORT3</td>
<td>67</td>
<td>11</td>
<td>78</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>ORT4</td>
<td>69</td>
<td>3</td>
<td>72</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>ORT5</td>
<td>67</td>
<td>12</td>
<td>79</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>ORT6</td>
<td>46</td>
<td>0</td>
<td>46</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>ORT7</td>
<td>52</td>
<td>17</td>
<td>69</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>PDL1</td>
<td>59</td>
<td>2</td>
<td>61</td>
<td>1</td>
<td>38</td>
</tr>
<tr>
<td>PDL2</td>
<td>54</td>
<td>2</td>
<td>56</td>
<td>32</td>
<td>12</td>
</tr>
<tr>
<td>PDL3</td>
<td>55</td>
<td>0</td>
<td>95</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>PDL4</td>
<td>45</td>
<td>28</td>
<td>73</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>PDL5</td>
<td>94</td>
<td>0</td>
<td>94</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>ENG1</td>
<td>74</td>
<td>12</td>
<td>86</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>ENG2</td>
<td>67</td>
<td>27</td>
<td>94</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>ENG3</td>
<td>97</td>
<td>1</td>
<td>98</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ENG4</td>
<td>30</td>
<td>5</td>
<td>35</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>ENG5</td>
<td>55</td>
<td>6</td>
<td>61</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>ENG6</td>
<td>53</td>
<td>3</td>
<td>56</td>
<td>9</td>
<td>35</td>
</tr>
</tbody>
</table>

The execution time for the 100-point test case (GO MUS), and the program size in bytes (Size).

The output produced for each of the 100 input data points was classified as "good" if the output was completely correct or was logically correct with "cosmetic" errors. The numerous cosmetic errors were due mainly to misspelling and bad output formatting. Other data points were classified as either detected or undetected error points. A point was considered to be a detected error if the program version caused execution of the instrumented code that had been added to detect and attempt recovery from abort conditions. In the far more serious case that the output looked legal but was in fact wrong, the point was considered an undetected error detectable only by external means. Table II shows the results of this classification.

Next, all possible triple combinations of the 18 versions were executed as an N-version module. Table III lists the breakdown of these 816 combinations. There were now three output points to consider for each input point, with the output points coded as in Table IV. Note that U is an undetected error that is not duplicated in one of the other two versions, U* an undetected error that is common to both or all three of the versions. The 14 meaningful combinations of these codes are shown with the corresponding voting function output in Table V. The distribution of the experimental results over the 14 voting categories is shown in Table VI.

All common errors were tabulated and traced to their causes. It was found that there were 21 different cases of common errors. Five of these were caused by specification limitations or errors, seven by logic errors made by the programmers, and nine by implementation errors. These common errors were tabulated in Tables VII-IX.

### WORK IN PROGRESS AND GOALS FOR LONG-RANGE RESEARCH

One major goal of the experiments described in the preceding sections is to apply the accumulated experience to the design of the next experiment. It has become evident that the general UCLA campus computing facility is an unsupportive and often hostile environment for multiversion software experiments. With a view to establishing a long-term research facility for such investigations, an effort is in progress to create a multichannel fault-tolerant system as an integral part of the...
UCLA Computer Science Department advanced local network facility, which uses the LOCUS distributed operating system. The projected six-year effort consists of four phases.

The first phase is the implementation of a multichannel fault-tolerant subset NIFTS, composed of at least three identical computing nodes (DEC VAX 11/750 computers) of the UCLA local network. It is to serve as an experimental vehicle for subsequent design fault tolerance studies. The SIFT $^2$ concept is being adapted at UCLA to serve as the foundation of NIFTS. In the second phase NIFTS will be used as the means to continue and expand the ongoing experimental research on the tolerance of software design faults that has been described in this paper. In the third phase we will investigate and implement a generalization of NIFTS to encompass N computing nodes with nonidentical hardware. Such an “N-fold diverse hardware” form of NIFTS is intended to tolerate faults due to left-over design errors and to errors introduced during modification and maintenance. In the fourth phase we plan to conduct extensive fault tolerance experiments with NIFTS as developed in the first three phases. The main goal is to evaluate the effectiveness and to refine the methodology of using N-version software and N-version hardware as mechanisms of design fault tolerance.

A second planned extension of our research is to employ the mail order concept of obtaining multiversion software. We are working to secure the cooperation of fault tolerance research groups at several universities in the USA and in Europe. Members of these groups will participate in writing the programs for a larger experiment that will be evaluated on our new experimental facility, NIFTS.

The practicality and generality of the design diversity approach as an alternative to fault avoidance remain to be established or disproved; however, the design fault problem in both software and VLSI circuits remains quite serious, and we consider our research results to be sufficiently encouraging to warrant further and more intensive efforts.

ACKNOWLEDGMENT

The research for this article was supported by NSF Grant No. MCS-78-18918 and by a research grant from The Battelle Institute. (Drs. Liming Chen and John P. J. Kelly have been major contributors to the research effort at UCLA.)

REFERENCES


