P-29

University of Illinois
at Urbana-Champaign

Campbell

Department of Computer Science
1304 West Springfield Avenue
Urbana
Illinois 61801
USA

NAG1-510

PRELIMINARY DESIGN OF

OF THE

REDUNDANT SOFTWARE EXPERIMENT

April 23, 1985

Roy Campbell
Lionel Deimel
Dave Eckhardt, Jr.
John Kelly
John Knight
Linda Lauterbach
Larry Lee
Dave McAllister
John McHugh

1. Introduction to the Experiment

1.1. Background

For some time, it has been suggested that the introduction of redundancy in software, in a fashion similar to that used in hardware, would increase reliability by providing fault tolerance. Under the assumption that software errors are randomly distributed through the replicate codes, very large gains in reliability are predicted. This assumption is equivalent to the random physical fault models on which hardware fault tolerance is based. While a substantial body of evidence exists to justify these assumptions for the hardware case, no firm evidence exists for the validity of the corresponding software assumption. In fact, the published accounts of experiments with fault tolerant software indicate that while fault tolerance does increase the reliability of software, the number of coincident errors among the replicate versions is greater than would occur if the faults were distributed in a random fashion.

It is fundamental to the continued development, and ultimate acceptance, of fault tolerant software techniques to arrive at an understanding of the nature and distribution of software faults and errors in order to evaluate the effectiveness of the strategy of redundant software. Recent analytical investigations [1] indicate that the redundant strategy may, in extreme cases, actually decrease the reliability of a system. However, in most cases, the strategy is effective although the reliability gain may be less than that predicted under the assumption of random faults.

1.2. Goals

The goal of the present experiment is to characterize the fault distributions of highly reliable software replicates, constructed using techniques and environments which are similar to those used in contemporary industrial software facilities. In order to achieve this goal, we will develop multiple copies of an application requiring several man-months of effort under rigidly defined practices involving design and code reviews, unit and system testing similar to those used in industry. The experiment will be governed by a carefully designed protocol and data will be gathered during the development process to assure compliance with the protocol and thus the integrity of the data gathered during subsequent life testing of the resulting software. The fault distributions and their effect on the reliability of fault tolerant configurations of the software will be determined through extensive life testing of the replicates against carefully constructed randomly generated test data. Each detected error will be carefully analyzed to provide insight into their nature and cause.

This and subsequent experiments will lead to an overall evaluation of the fault tolerant strategy. A direct objective is to develop techniques for reducing the intensity of coincident errors, thus increasing the reliability gain which can be achieved with fault tolerance. Data on the reliability gains realized, and the cost of the fault tolerant configurations can be used to design a companion experiment to determine the cost effectiveness of the fault tolerant strategy. Finally, the data and analysis produced by this experiment will be valuable to the software engineering community as a whole because it will provide a useful insight into the nature and cause of hard to find, subtle

faults which escape standard software engineering validation techniques and thus persist far into the software life cycle.

2. The Application

After an extensive search for an appropriate application from the avionics field, one has been found that will meet the established requirements and limitations. This application is the "Failed Sensor" problem originally suggested by Alper Caglayan of Charles River Analytics. In this application, outputs from eight linear accelerometers, each of which has a different orientation, are processed to produce the three primary axis accelerations. Outputs from four two-degree-of-freedom rate gyros, each of which also has a different orientation, are processed to produce estimates of the angular body rate of the aircraft. Account must be taken for the possibility that accelerometers and/or gyros fail; failed sensors must be detected, and their outputs must be excluded from the computations.

There are (at least) three algorithms for detecting the failed sensors. We will provide the students with general functional specifications and a description of one of those methods, the Parity Method.

One criterion for an appropriate application is that it require over 1000 lines of code; a reasonable implementation of the basic problem would result in less than 1000 lines of code, but the basic problem can be enhanced to meet the requirement and provide additional (and realistic) complexity.

Another criterion for the chosen application was that it be easily enlarged, in case students complete the assigned work faster than is anticipated. The "Failed Sensor" application can easily be lengthened in mid-experiment by adding requirements such as having the code turn on and off LEDs, driving a seven-digit BCD display, etc.

3. The Computing Environment

3.1. Introduction

Software may be greatly influenced by the environment in which it is developed and the environment for which it is intended. In order to eliminate this possible source of variation between the software produced by the programming teams, the experiment will be conducted using a common development environment and a common acceptance and evaluation environment. Since the emphasis of the experiment is on producing reliable software, the production tools and development environment should be of "industrial quality". The acceptance and evaluation environment may be the same as the development environment, but it may be desirable choose an environment that would allow testing to occur on a supercomputer. This section defines the development environment of the project and outlines the facilities required in the acceptance and evaluation environment.

3.2. Development Environment

3.2.1. Operating System

The Berkeley UNIX 4.2BSD system has been chosen for the development environment because it is widely available, it is used in industry to support software production, and it offers many software tools in a sophisticated environment. Because some 4.2 systems may have floating point units and others may not, the development environment should use floating point emulation. Because some implementations of Berkeley compilers on different hardware may lead to different output on different machines, we propose using the VAX 750 implementation of Berkeley 4.2. The target environment will have (does the CRAY have ISO?) hardware floating point.

3.2.2. User Interface

For its flexibility, history facilities, and simpler shell script syntax, the C shell will be used for the user interface of the development environment.

3.2.3. Protection

Each team will have its own separate group 'universityid[A-H]' and software developed by that group *must* be stored under a directory with only 'universityid[A-Z]' group read and write access. The university identifier for a University is its ARPA net or CS net address (e.g. 'Uiuc' for the University of Illinois.) No member of any team should have supervisor privileges.

3.2.4. Use of File System

Software should be stored in a directory hierarchy, using the file system to support the software structure. Each directory should include a makefile for the software contained in that directory and a README file that documents the software structure represented by the directory. All code should be labelled using the dot convention: '.h' for header information, '.i' for include files, '.p' for Pascal, '.o' for object code, and '.t' for text processing source. Symbolic links may be used if required. Normal links should not be made to within the team directory.

3.2.5. Submission of Software for Testing

Software will be transmitted to the software testing site by UUCP or FTP file transfer. These file transfer facilities provide better reliability for the transmission of files than mail and allow all characters to be correctly sent. The test site machine will initiate all collections of software except over Arpanet. A directory, '/usr/spool/ft-expt/universityid/groupid' will be available on the test site machine to receive software to be tested. For security, the principal investigator at each university site must initiate the software transmission by UUCP/FTP. Students participating in the programming must not have access to the test files on the machine used for testing. The software to be transmitted should be structured within a directory and this base directory should be named 'base.version-number'. The directory should contain a makefile that will construct an object file called 'system' that is located in base: 'base/system'. Software should be transmitted in 'tar' format and should include the base directory 'tar -r

tosendfile base/*'.

3.2.6. Collection of Test Reports

The result of a test will be returned via 'mail' to the local coordinator.

3.2.7. Other Communications

Except for very specific purposes, all other communications will be made by mail.

3.2.8. Testing

The teams will be provided with a simple testing harness constructed using C shell scripts and makefiles.

3.2.9. Tools Set

A complete tape of the tools recommended can be made available to schools if they do not already have them. The standard tool set is described next.

3.2.9.1. Pascal

Berkeley Pascal (PC) will be used for program development. The ISO Pascal standard should be adhered to as a coding practice. (The standards will be needed more to ensure protability than code quality.) No UNIX specific extensions may be used by the developers except for separate compilation. All input and output will be performed by invocations of a suite of I/O routines supplied with the testing tools. The separate compilation features of Berkeley Pascal can be used by the teams to simplify their work.

3.2.9.2. Editor

Although it would be good to allow several different editors, for uniformity we will use the screen-based text editor 'vi'.

3.2.9.3. SDB

Symbolic debugging of programs may be performed using SDB although there are some problems with this debugger. DBX only partially works for Pascal but could also be used.

3.2.9.4. Version Control

RCS will be used for version control because it provides fast retrieval of the current version. Every separate file storing a component of the software should be archived by RCS with a separate name and version number. Logging should be used and the log file kept up to date. Automatic version numbers should be maintained and these numbers should be included in the text of the program, as a character array constant in the object code produced for that text, and as text output of the program. The authors of the program should likewise include their name in the text, object code, and output.

3.2.9.5. Pascal Cross Reference and Pretty Printing

A Pascal Cross Reference option is provided by the program "pxp" and should be used to produce cross reference listings for the purposes of development. The program "pxp" can be used to remove include files and header files and produce a single Pascal

program listing. The program may also be used to pretty print the Pascal. This would appear necessary if the programs are to be compiled for the CRAY.

3.2.9.6. Gprof: Profile program

Gprof may be used to obtain a run-time execution profile of a Pascal program.

3.2.9.7. Configuration Control

Makefile scripts should be used to support configuration control for ease of testing and compilation.

3.2.9.8. Documentation Tools

Documentation techniques should be similar for every project. All text processing associated with documentation should be accomplished using the me macros and nroff, troff, ditroff text processing systems. Tables should be prepared using 'tbl', equations written using 'eqn', and pictures drawn with 'pic'.

3.2.10. Training

To ensure uniform skills amongst the development teams, we propose a 1 week exercise in which we bring all participants up to the required level of knowledge and familiarity with the software tools available and the software engineering techniques and protocols proposed.

3.3. Evaluation Environment

3.3.1. Acceptance Testing

We propose that acceptance testing be conducted at a centralized site for all schools. The advantages of a centralized site are:

(1) we may use a CRAY or some other fast processor
(2) not all sites need be concerned with the mechanics of testing (other sites may be involved in generating the test cases)
(3) the need to distribute test cases is eliminated (though at the cost of transmitting the programs and test results)
(4) the testing records can be maintained at one site
(5) uniformity of testing can be ensured
(6) if any changes are made in the testing procedures during the experiment, they are more easily applied in a uniform manner.

3.3.2. Evaluation Testing

The high reliability expected of the programs indicates that the evaluation testing of the software will involve a very large number of trials. Therefore, we propose using a CRAY XMP supercomputer to accomplish this task. The Cray Pascal must be compatible with the Pascal provided by the VAX. Cray Pascal is compatible with the ISO Pascal standard but has some restrictions and some extensions. The major restriction is that it requires all lines to be less than 140 characters long.

The Pascal programs will be collected on a VAX as single programs, copied onto a tape, and transferred to the CRAY. Diagnostics from the CRAY will be returned by tape to the VAX and will then be distributed in the form of test reports.

4. Development Methodology

4.1. Introduction

Development methodology refers to the software development methodology employed by the programmers during the software development process. In a sense, the development process does not matter a great deal. Whatever results are achieved by this experiment, they will be *conditional* on the development process. Thus any development process would, in principle, be satisfactory. However, if the results are to be believed and regarded as useful by industry, we should adopt a development approach that resembles as closely as possible the methods used by industry. In this experiment, our potential number of versions is already very low and so we had better ensure that every version we pay for is acceptable for analysis.

The development process is influenced by the students' backgrounds. Can we require that they have all had specific course work? Can we assume they all understand major topics such as abstract data types or structured design? Probably not, and even if we could, there would be other technologies that we would like to use but which are insufficiently known. Differing educational backgrounds is an awkward problem. The solution discussed informally at various meetings is threefold:

(1) Provide each student with a copy of a standard text (Fairley's has been suggested) and require that they read it at the beginning of the experiment.

(2) Run a five day training seminar at the beginning of the project. (Does everyone agree that the training should take five days?)

(3) Stop worrying about the problem and assume diverse ability contributes to design diversity.

Since the programmers will be supplied with requirements specification documents, we are spared the requirements analysis and the preparation of the requirements specification stages of software development. Also, we assume there will be no post-delivery enhancement or fault correction, so there will be no need to consider the phase euphemistically known as "maintenance". Thus, we suggest that development needs to include design, code development, and validation only. For the purposes of discussion, we propose the methodology outlined in the next section and the protocol outlined in the section three.

4.2. Background And Development Logging

We need to know who our programmers are. They should fill in a questionnaire detailing their backgrounds. We need to know exactly what is being done when. We propose, therefore, that we require a work log be maintained, in which each work period is documented. (It was hoped that some of this logging could be done automatically, but there is not enough time to implement the program for this experiment.)

4.3. Specifications

The experimenters will provide a complete high-level external specification. All input and output will be defined through a set of parameters that the program version will use.

At all stages, questions about the specifications will be submitted to the RTI Coordinator by electronic mail, reviewed and responded to by electronic mail All questions and all responses will be broadcast by the Coordinator to all programmers at all sites, and will be logged for future reference by RTI as well (see Protocol, section 5.2).

4.4. Design

We propose using ad hoc design using information hiding and abstract types only. The design will be documented in a form yet to be specified and be delivered on a specified date. A design walkthrough will be required involving only the development team and a report to be produced of the results of the walkthrough. This, and in fact all other walkthroughs, will be attended by the experimenter and/or an aide but with silent participation.

The first deliverable item will be a design document. The content will be a diagram showing the abstract data types and abstraction layers that the team intends to use, a listing including the major data types and variables that the program will use, expressed in Pascal VAR and TYPE parts, the headers of all the procedures that the

program will use including the specification on all the parameters, and a comment explaining the procedures purpose. This document will be due on a date yet to be specified.

4.5. Code Development

Code development will be done in Pascal using coding standards provided by the experimenters. The code will be developed up to system compilation only, i.e. there will be no "random" executions of the entire program. Unit testing will be performed on the individual parts as they are written. Code walkthrough will be required involving only the development team and a silent observer, and a report will be produced of the results of the walkthrough.

The program will be developed in a strict top-down fashion in which each layer of the abstraction will be implemented and tested as a unit using stubs for the incomplete lower layers. The second deliverable will be a series of compiled programs representing the results of the top down development at each abstraction layer. Testing of each layer will be by a small number of ad hoc tests that the team deems suitable. The team will be responsible for developing the necessary test drivers. These tests will be aimed at removing the major flaws in the layer only. Once the entire source text has been integrated, the program will be validated according the test plan.

4.6. Validation

A test plan and test log will be required, with both to be documented and delivered on a specified date. The validation will be performed by testing only, and will be limited to functional testing.

Each team will develop test drivers to assist in the test process for each of the three test phases, but again these are to be the only software tools used in validation. All test executions during validation must be logged; the completed log is the fourth deliverable item. The fifth deliverable is the final program.

4.7. Acceptance Testing

Acceptance testing is our determination of whether the software is of adequate quality to be used in the experiment. The specification of the form of the acceptance test is not part of the development process. The action to be taken following failure is. Naturally, we require that the delivered software satisfy the acceptance test at the end of the development process. In the event of failure, we propose that the programmer be required to document his actions in his development log *in detail*; every design change, every changed line of code, every recompilation, every re-executed test. Programmers will be provided with a standardized method for tracking code changes (see Protocol, section 5.6). We also require that the programmers keep trying until they have passed the acceptance test, no matter how long it takes.

5. Protocol

Protocol covers the rules and guidelines to be followed by the experimenters and the programmers during the experiment. Unlike development methodology, protocol is *crucial;* if the development protocol fails in some way, for example if we cannot guarantee that we have preserved independence during development, or versions are not completed on time, the entire experiment will have been *wasted.*

This section sets forth the protocol from the hiring through the acceptance test phases of this experiment. Any problems occuring during the experiment that are not covered in this section should be handled by sending electronic mail (or calling, depending upon the urgency of the problem) to the RTI Coordinator; the Coordinator will help work out a solution, and log the problem and its solution. This will ensure all similar problems are handled in the same manner across Universities, and will result in one comprehensive log of all problems encountered during the coonduct of the experiment.

5.1. Recruitment

Six graduate and/or qualified undergraduate Computer Science students will be recruited at each University. (Funds and quantity of qualified applicants permitting, more than six may be hired.) The employment advertisement and application form prepared by John Knight should be used at each of the four universities, so we will have a standard by which to compare applicants, not only within site but across sites.

Applicants will be hired for a ten week period. Working hours for the programmers will be flexible, but at least forty hours per week of effort is required.

A homogeneous group of experienced programmers is desired. Qualities of successful applicants include: experience coding longer (over 500 lines) programs, Pascal and Unix experience, C.S. work experience, good grades in a variety of C.S./Math courses, and a reputation as a motivated,diligent worker.

Immediately upon being hired, the programmers will be given a questionnaire asking for elaborate background information. Although this data would not likely be used in this task, it will be available for use in future studies involving the data collected in this experiment.

If an experimenter expects to be gone for a significant amount of time during the conduct of the experiment at his University, he should also hire an Aide, to take care of administrative duties in his absence. (This employee could be hired for half-time work.)

5.2. Training

In an effort to avoid what T. X. Barber [2] calls the "Investigator Loose Procedure Effect" and defines as the "degree of imprecision of the experimental script or protocol which gives the step-by-step details of the procedures to be used in the experiment", most of the programmer training will be with written materials. The only experiment-related verbal communication between experimenter and programmers should be the initial experiment overview presented to the programmers by the experimenter. This

overview will consist of three presentations:

(1) Experiment overview, ground rules, and schedule

(2) Software tools and facilities to be used

(3) The application and documents to be prepared by the teams

A standardized written outline (and overhead projector slides) of the contents of these presentations will be developed, and must be followed closely by all experimenters. RTI will develop the outline for all subjects with the exception of site-specific facilities. Each experimenter is responsible for developing an outline (and handouts) covering location and operation of terminals and printers, and the site logon/logoff sequence.

No questions (with the exception of site-specific questions raised during the discussion of local facilities) will be allowed during these presentations; programmers will be told at the beginnning of the presentations to write down any questions, so they can later mail them to the experimenter (or his Aide), or the RTI Coordinator, as explained below.

Site-specific questions the programmers have may be mailed to the experimenter (or Aide); these include questions concerning onsite hardware, lost documents, etc. The experimenter will send both the question and its answer to all programmers at his site. All other questions/comments the programmers have are to be mailed to the RTI Coordinator. The coordinator will answer the question (after consulting over the phone with the other experimenters if necessary) and mail both the question and its response to all programmers and experimenters. The Coordinator will keep a log of all questions

received and answers supplied.

Timeliness is very importance in response to all mail received; a student at times may feel he can not continue his work until he has the answer to a question. If at any time a student finds himself or herself in this position, and has waited 24 (48?) hours for a response, he or she should call the RTI Coordinator to see that they received the message and to obtain a response.

The verbal overview, handout with all due dates, and all other written materials *except* the experiment application specifications will be given to the students on their first day of work. The first five working days will be allowed for familiarization with the programming environment. During this time, students are to read all training materials, become familiar with the tools available to them, and complete a training exercise which requires similar skills to those needed for the experimental application. The specifications for the training exercise are a subset of those used in an experiment conducted by Nagel and Skrivan [3]; the original specifications call for calculation, given the longitude and latitude of two points, of:

(1) the great circle distance between the two points (in nautical miles)

(2) the azimuth of the path from the first to the second (in radians), and

(3) all intersections (if any), listed in the order encountered as the path is traversed from point 1 to point 2, of the great circle path connecting points 1 and 2 and the small circle defined by point 3.

To insure that even those programmers who must spend a significant portion of the week familiarizing themselves with the computing environment can complete the training exercise, only parts (1) and (2) {or (2) and (3)} of the original specifications will be assigned.

The training period and the training exercise are an effort to ensure programmers are not still *learning* after the training phase, as this could affect resulting code quality. Programmers may ask each other any Pascal, Unix, or site-specific questions during this phase. However, they must still converse with the experimenter through electronic mail. Although this means a difficult change in communication policy for the programmers once the training period is over, their ability to openly communicate among themselves during this phase will increase their resources for learning the experiment environment.

All programmers should turn in the completed training exercise at the start of the sixth working day. This is for proof of effort only; it is not intended that this become a condition of employment. (However, it should be decided *now* what, if any, action will be taken if any student does not complete this exercise.)

5.3. Team Assignments

Students will be ranked in ability based on information in their application forms and previous experience. To simulate a senior/junior pairing in an industrial environment, teams of two should be formed by grouping those individuals rated 1&6,

2&5, and 3&4.

The experimenter should meet a for short time with the entire group of programmers early on the sixth working day, to announce his team assignments and hand out the program specifications. At this time he should also reiterate the importance of independent development by teams, and that all questions will be handled by electronic mail.

5.4. Design Phase

During this and subsequent phases, no verbal communication directly concerning this experiment will be allowed between experimenter and student or across teams. (Experimenters may of course talk with students about subjects other than those relating to the experiment.) The experimenter will receive evidence of progress by the documents received from teams at each stage of study, and an 'agenda' handout as well as an online calendar file will be sufficient to remind programmers of all due dates.

We need to decide on the format and contents of the documents/work logs to be received from the programmers, and set up sufficient deadlines for handing these in.

5.5. Coding Phase

This phase includes coding, walkthroughs, and unit testing. At the beginning of this phase, the two members of each team are to decide who will code and who will test which units; each programmer should end up coding approximately 50% and unit testing 50% of the time, and a unit coded by one person must be unit tested by the other person. As mentioned earlier, documents will be handed in at every phase; this should ensure students meet deadlines and eliminate the need for verbal communication between experimenter and students concerning progress. Documents to be turned in during the coding phase will include list of coding/unit testing breakdown between team members, the compiled listings of the application, a walkthrough document, a time log, and a unit test error log.

5.6. Integration Testing Phase

The number of test cases executed must be logged. In addition, for each test failure, we will need to receive, at a minimum, the following information: input revealing error, error output, error type, fix, explanation of fix. Fixes will be tracked in source programs by a method of descriptively numbering fixes and surrounding the modified code with its fix number and optionally other comments. The detailed fix procedure will be explained in a handout.

5.7. Acceptance Test

We must require that all integration testing be done by a reasonable amount of time before the end of employment. The length of time allowed for acceptance testing is very important, as we will have to ship the programs to the CRAY for testing and return to programmers; all this takes time, and we don't want our students leaving before their team's program has passed the acceptance test!

The acceptance test will consist of 500 test cases, randomly generated within the problem domain. (500 test cases will be randomly generated for each program; therefore some inputs may be the same across some test sets, but each test set will not be identical.) Those programs that do not pass all tests in their test set will be returned, along with a record of the test inputs for which that program failed. If a program is 'fixed', and does not pass when subjected to the *same* test set, it is again returned along with those inputs for which it appears to fail.

It has been proposed that we offer a sliding bonus to students, depending on how quickly their program passes the acceptance test. (This is meant to inspire the students to produce reliable code; this idea is open for discussion.) Under this plan, if a program passes the acceptance test on the first try, the students who created it receive 100% of the bonus; if the program passes on the second try, they will receive, say, 90% of the bonus...etc. We need to set a maximum dollar amount on the bonus, and a ceiling on the number of acceptance tests required, for which program creators will still receive any bonus, if we go with this plan.

5.8. Post Experiment Questioning

It has been proposed that on the students' last day on the job, they be given a post-experiment questionnaire. Questions asked could include, for example:

1. Did you have or notice any application-dependent conversations across team boundaries? If so, about how many times, and concerning what?

2. Did you learn anything about the tools/envi. after the training period? If so, what?

3. How hard was the application? 1(simple) -- 5(very difficult)

4. For each phase of this experiment (training, design, coding & unit test, acceptance test) please comment on how well matched the amount of work was to the time allowed for completion of the work.

5. How do you think you and your partner compared as far as skill level goes? (Only consider skills needed for all phases of your employment this summer.) Use the scale 1(almost equal skill levels) -- 5(extremely different skill levels)

6. Estimate the percentages of the total work you and your partner did.
   You:
   Partner:
   (total = 100%)

7. Rate your record keeping on a scale of 1 (extremely accurate) -- 5(extremely inaccurate)

8. Did you use any references in the course of the summer? If so, please specify titles and type of information referenced. (Do not include provided handouts, but do include texts used from the provided reference list as well as texts/articles not on the list.)

9. What, if anything, would you do differently if you were designing a similar experiment in the future?

6. Issues To Resolve

Here is a list of issues in the development process and protocol areas that we need to discuss at the next meeting. Of course, everybody is encouraged to add to this list as they see fit.

(1) What procedures are we going to follow and what rules are we going to enforce to maintain development independence?

(2) In what form should the documentation we require be presented? If we determine that there are flaws in a particular part of the development (for example, a design is inadequate) should we do anything to correct the situation? In a practical environment, the programmers would be faced with management and customer reviews as they went along. Do we want to try to model this? Should we develop a checklist to judge design documents by, and return the document and checklist if problems exist?

(3) What questions do we put in the background questionnaire?

(4) What form should the development log take? How do we ensure it is kept accurately? Do we really care or need it (of course we do)?

(5) What detailed restrictions on language elements should be imposed? This is most important if we are going to ensure portability to many machines for testing.

(6) Should any other software tools be used, required, permitted? If so, which other tools?

(7) What approach should be used in synchronizing events to ensure all the teams work at roughly the same rate and that deliverables are available on time?

(8) Design is of course an iterative process, and as such we could require more than 1 design document from students during this phase. This would have the benefits of giving students regular deadlines to meet, and giving experimenters assurance of students' progress. This will require that we come up with formats for students to follow in each progressive document, though.

(9) We need to decide upon the format and contents of the "proposed test strategy" document students should turn in with the final design document.

(10) Should we provide a bonus for good design? (If so, how do we judge?)

(11) Should we provide a sliding bonus for time taken to pass the acceptance test, as described in section 5.7?

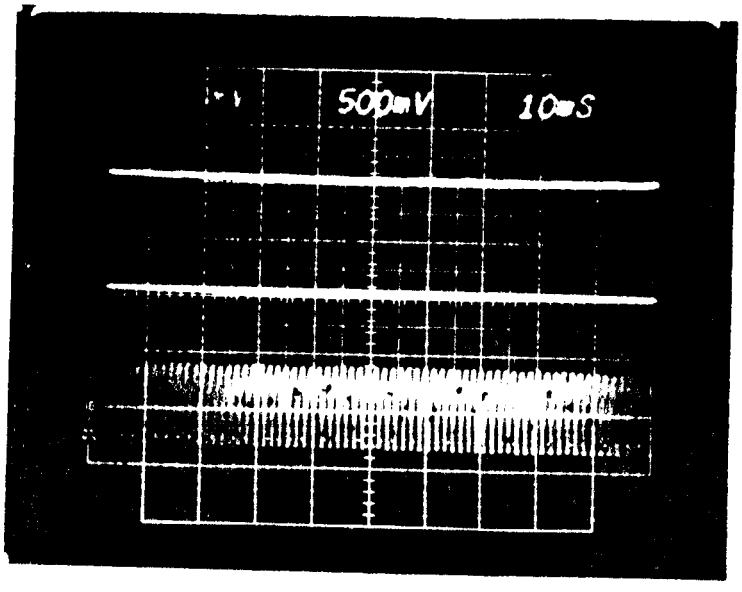
(12) What is an appropriate time schedule of events and document deadlines for this experiment?

References

[1] D. E. Eckhardt, Jr., and L. D. Lee, "A Theoretical Basis for the Analysis of Redundant Software Subject to Coincident Errors", NASA Technical Memorandum 86369, January 1985.

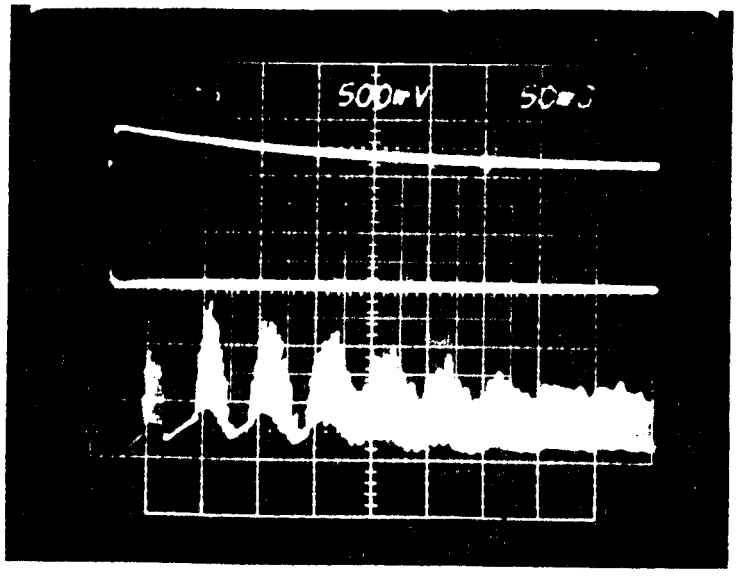
[2] T. X. Barber, *Pitfalls in Human Research: Ten Pivotal Points*, Pergamon Press, Inc., 1976.

[3] P. M. Nagel and J. A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling", Boeing Computer Services Company report, NASA Contract NAS1-16481, February 1982.



— CO_2 Discharge Laser Output

FIG. 4
(Before blackbody discharge)



— Blackbody Radiation Intensity

— Discharge Current(Blackbody)

— CO_2 Discharge Laser Output

FIG. 5
(After blackbody discharge)