[5] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *Comput. Surveys*, vol. 10, pp. 123–165, June 1978.

[6] D. L. Parnas, "Information distribution aspects of design methodology," in *Information Processing 71*. Amsterdam, The Netherlands: North-Holland, 1971, pp. 339–344.

[7] R. E. Frankel and S. W. Smoliar, "Beyond register transfer: An algebraic approach for architecture description," in *Proc. 4th Int. Symp. Comput. Hardware Description Languages*, Oct. 1979, pp. 1–5.

[8] C. G. Davis and C. R. Vick, "The software development system," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 69–84, Jan. 1977.

[9] E. R. Buley, R. E. Frankel, and S. W. Smoliar, "Data processing system modeling: A process-oriented approach," in *Proc. 11th Annu. Modeling Simulation Conf.*, May 1980, pp. 675–679.

[10] S. W. Smoliar, "Simulating distributed systems: A two-level approach," in *Proc. AIAA Comput. in Aerosp. III*, Oct. 1981, to be published.

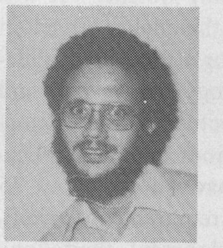
[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing 74*. Amsterdam, The Netherlands: North-Holland, 1974, pp. 471–475.

[12] D. M. Andrews, "Using executable assertions for testing and fault tolerance," in *Proc. 9th Annu. Int. Symp. Fault-Tolerant Comput.*, June 1979, pp. 102–105.

[13] D. F. Palmer *et al.*, *Distributed Data Processing (DDP) Technology Program Operational Requirements Accommodation: First Quarterly CY 80 Engineering Design Notebook*, General Res. Corp., Tech. Rep. CR-12-825, BMDATC Contract DASG60-78-C-0034, Apr. 1980.

[14] J. H. Wensley *et al.*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 68, pp. 1240–1255, Oct. 1978.

[15] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 613–641, Aug. 1978.



Stephen W. Smoliar (M'76) received the S.B. degree in mathematics and the Ph.D. degree in applied mathematics from the Massachusetts Institute of Technology, Cambridge, in 1967 and 1971, respectively.

From 1971 to 1973 he served as Instructor in the Department of Computer Science, Technion–Israel Institute of Technology, Haifa. From 1973 to 1978 he was Assistant Professor of Computer and Information Science at the University of Pennsylvania, Philadephia. At this time he was cosupervisor of a research project concerned with computer-assisted preparation and interpretation of dance notation scores. From 1978 to 1981 he was a member of the Technical Staff of the Santa Barbara Division of General Research Corporation. Currently, he is with Schlumberger-Doll Research, Ridgefield, CT. His research has been primarily concerned with distributed data processing requirements engineering and modeling and simulation of distributed systems for military applications. He has also published extensively in the areas of music theory and dance criticism.

Dr. Smoliar is a member of the Association for Computing Machinery and Sigma Xi.

Application of a Methodology for the Development and Validation of Reliable Process Control Software

C. V. RAMAMOORTHY, FELLOW, IEEE, YU-KING R. MOK, MEMBER, IEEE, FAROKH B. BASTANI, GENE H. CHIN, MEMBER, IEEE, AND KEIICHI SUZUKI, MEMBER, IEEE

Abstract—**This paper discusses the necessity of a good methodology for the development of reliable software, especially with respect to the final software validation and testing activities. A formal specification development and validation methodology is proposed. This methodology has been applied to the development and validation of a pilot software, incorporating typical features of critical software for nuclear power plant safety protection. The main features of the approach include the use of a formal specification language and the independent development of two sets of specifications. Analyses on the specifications consists of three-parts: validation against the functional requirements consistency and integrity of the specifications, and dual specification comparison based on a high-level symbolic execution technique. Dual design, implementation, and testing are performed. Automated tools to facilitate the validation and testing activities are developed to support the methodology. These includes the symbolic executor and test data generator/dual program monitor system. The experiences of applying the methodology to the pilot software are discussed, and the impact on the quality of the software is assessed.**

Index Terms—**Assertion, dual-programming, methodology, path analysis, process control, reliability, requirement, specification, symbolic execution, testing, validation, verification.**

Manuscript received July 22, 1981.

C. V. Ramamoorthy is with the Computer Science Division and Electronics Research Laboratory, University of California, Berkeley, CA 94720.

Y.-K. R. Mok was with the Computer Science Division and Electronics Research Laboratory, University of California, Berkeley, CA 94720. He is now with Bell Laboratories, Naperville, IL 60566.

F. B. Bastani was with the Computer Science Division and Electronics Research Laboratory, University of California, Berkeley, CA 94720. He is now with the Department of Computer Science, University of Houston, Houston, TX 77004.

G. H. Chin was with the Computer Science Division and Electronics Research Laboratory, University of California, Berkeley, CA 94720. He is now with the Information Networks Division, Hewlett-Packard Corporation, Cupertino, CA.

K. Suzuki was with the Computer Science Division and Electronics Research Laboratory, University of California, Berkeley, CA 94720. He is now with Intel Corporation, Santa Clara, CA.

I. Introduction

THE NEED for a methodology to develop reliable computer software is becoming increasingly important to the nuclear industry as the role of digital computers in the operation, control, and safety of a nuclear power plant expands. In these critical applications it is important that the system will behave as expected for all possible demands and input conditions. Emphasis is placed on preventing errors in performance when considering safety and cost.

This paper discusses the evolution, design, and implementation of a practical top-down software development methodology for nuclear reactor safety (protection) systems. It was developed with a tremendous concern for the quality (freedom from errors) as well as effort required for a thorough validation. Since its inception the methodology was revised and improved three times. The flexible structure of the original methodology enabled these revisions without major complications. Since the previous studies [30] have shown about 75 percent of errors occur in the front end (requirements specification and design) phases of software development, the following features were incorporated.

The functional (originating) requirements developed by nuclear engineers were analyzed independently by the two development teams (Babcock & Wilcox and University of California, Berkeley) and with the help of a third team (Science Applications, Inc.) they were revised. Then the two development teams derived specifications (also called preliminary designs) in a formal language RSL [1]. Each team manually verified the transformation against the originating requirements individually. The two "validated" specifications (preliminary designs) were compared against each other so that any ambiguities, misinterpretations, etc. could be detected. After each preliminary design had been revised, it was ready for back-end phases such as detailed design, implementation, testing, and verification.

Major problems incurred in the implementation and testing phases are the cost of detection, location, and correction of errors. Since in our project the computations were highly complex, it was very difficult to develop correct results for a specified set of inputs. Traditionally, most of the cost incurred in the back-end software development phases are in developing test cases (inputs) whose outputs are known and in locating the cause of the error, when detected. In our project, the computations were not only complex but also our inputs were from physical sensors (temperature, pressure, neutron flux, etc.) which changed gradually over time and were subjected to noise. To overcome these problems, each team (B & W and Berkeley) developed detail designs and implementations from their individual revised and corrected specifications, and subjected each program to walk-throughs, static and dynamic analysis, and testing. After each team is satisfied with its implemented program, the execution results of the two programs were compared with each other against identical inputs. If the results are the same or within some specified numerical threshold, then it is assumed that the result is correct; otherwise one or both programs are incorrect for that specific input set. In the latter case, both programs are analyzed against the originating requirements, to obtain the reason for deviation of the result. The program associated with the error is then identified and corrected. We used an automatic test data generator to create inputs and a dual program monitor and analyzer to exercise the two programs to help identify errors contained in them. Several thousand test cases can be generated to automatically test the programs so that errors can be detected and corrected. Our experience indicates that savings in testing, error detection and location far out weighed the cost of dual (two independent) program development. The size of the programs were about 1900 lines of code in structured Fortran (Iftran). The time to implement (coding and debugging) was only about six weeks. It took about three and a half years to develop the software methodology and the automated tools. The latter are one-time costs. It is our belief the methodology and its tools are similar to the manufacturing aids of an automobile assembly line. Since nuclear reactor safety systems form a family of related systems it would be possible to implement safety system software for a wide variety of reactors using these generic automated aids rapidly.

One of the programs (from Berkeley) after being thoroughly tested (and debugged) was subjected to formal validation. This was done in two phases. In the first phase, the input and output assertion to each ALPHA (RSL term for a module or a function) was developed. Then the RSL program of the safety system was subjected to high-level symbolic validation. In other words, the output assertion of the *program* is checked against the input assertion of the *program* and the high-level path conditions of its modules. The latter is derived from the I/O assertions of the ALPHA's. Since the number of paths at the level of specifications is small, this high-level symbolic validation was feasible. In this way a formal validation at the level of specification was performed.

In the second phase each module or ALPHA is validated similarly using symbolic execution at Fortran level. Since we are not using the whole program but a single module, the number of executable paths were generally small. Thus each ALPHA is validated against its I/O assertion. Thus the two phases provided a two-level formal validation of our program against the formal specifications.

It was also required to assess the quality of the software by means of software reliability models. In a process control application, operational reliability of the integrated hardware-software system is desired. Since the determination of hardware reliability is well known, one needs an appropriate software reliability measure to develop the operational reliability of the total computer system. One phase of our project consisted of gathering error data and developing software reliability measures.

We also incorporated the following features in our methodology.

- Each step in the development is derived from the previous step by formal procedures.
- Automated tools are used to validate each step against the preceding step (from which it is derived).
- Systematic derivation of test cases from each step of the software development.

The methodology has been applied to the development of a pilot software to compute the core thermal power in a nuclear reactor protection (safety) system. The nuclear plant safety system hardware consists of channels and sensors. The sensors monitor specific plant parameters (e.g., neutron flux, pressure, temperature) and each channel generates a trip signal when parameters from sensors exceed predefined limits. The system generally uses quadredundant channels and initiates protection actions on the basis of two-out-of-four logic. Digital computers are incorporated in each channel to compute the trip signal. (More details and references are found in [11].) The functional requirements have been selected to emphasize features which would be typical in nuclear reactor applications. These include the following:

- a large amount of computation,
- a relatively simple control structure,
- critical timing requirements, and
- computation based on past history of measured data.

The nuclear reactor safety system is an integral part of all nuclear reactor systems. The safety system functions for nuclear power plant applications are quite distinct from system to system. However, there are several common functions. The modules in the system are reusable, easily modifiable, and can be added or deleted according to requirements changes. One characteristic of nuclear reactor safety system software (and other process control software) is that the software engineers entrusted with the development task are not completely familiar with the functional requirements. This is unlike most scientific programs, such as sorting and matrix multiplication, in which the functional requirement is well defined and interpreted correctly by the software designer. Thus one major aspect in our methodology is to require formal validation of preliminary design against the functional requirements to detect and eliminate most errors that are due to ambiguities and misinterpretations as early as possible.

In this paper we discuss the methodology and evaluate it based on our experiences. In Section III, the requirements specification and the preliminary design of the application program are described. The detailed design and implementation are discussed in Section IV. Section V discusses the problems of validation and testing together with our experiences in this application. Section VI discusses the reliability assessment techniques. The various automated tools used in the project are described as needed.

This paper is admittedly a summary of a very broad research project. The research has generated a vast amount of literature on all aspects of the methodology. These are referenced wherever appropriate.

The project was initiated and conducted under the auspices of the Electric Power Research Institute. Babcock and Wilcox, and U. C. Berkeley were the two teams responsible for the design and development of the software system. The algorithms used in the functional requirements were selected jointly by the Electric Power Research Institute (under Dr. A. B. Long) and Babcock and Wilcox (Development Team 1). University of California at Berkeley (Development Team 2) had the overall responsibility for the methodology and tool development and was responsible for the verification and validation activities. Science Applications, Inc. and General Research Corporation were the independent evaluators and reviewers of the project and were responsible for the documentation and validation.

II. A Methodology for the Specification and Validation of Critical Software

Since the inception of the project, the software development and validation methodology have been revised three times. The flow diagram of the final version of the methodology for the nuclear power plant protection software is shown in Fig. 1. The major features are as follows.

1) The engagement of two separate independent development teams to produce two sets of software from the same requirements (for various purposes discussed later).

2) The use of a third independent team to coordinate the activities and to test, review, and intercompare the intermediate and final results.

3) The specification of software functions is documented by a formal specification language that allows machine analysis and rigorous comparison of the two specifications.

4) Validation of the final software is achieved through a number of steps throughout the system development, i.e., the program is tested and proved against the software specification, and the specification is validated against the functional requirements, and the functional requirements are analyzed in several levels of decomposition. Tools used for testing and validation includes the automated test data generator and dual program comparator for representative testing, and the symbolic executor for validation.

5) Integration testing is performed by all three teams.

6) A quantitative assessment of correctness and reliability is made using representative testing and symbolic execution.

The methodology has been described in more detail elsewhere [11]. The main differences between this final version of the methodology and previous versions are 1) the addition of a third team to perform independent validation, verification, and documentation of the software and 2) the performance of dual programming on all critical and noncritical paths. These provide added confidence and a separate view of the finished software. Two sets of specifications are developed from the same functional requirements. This allows us to catch most of the ambiguities in the functional requirements. Implementation of two programs from two independently developed versions of specifications allow the use of dual comparison for testing and validation. This is one reason for the adoption of the dual development approach in the methodology. One major characteristic of the nuclear reactor safety protection software is the time correlation of the program inputs; this made the task of determining the correct output for a particular set of test inputs almost impossible. The use of dual development and testing thus eliminates the need for determining the correct system response *a priori*, and allows for execution of a large number of test cases since it is unlikely that the two programs will contain identical errors. Also, by comparing intermediate outputs errors can be more easily located. One question that often arises is why the magic number two, why two sets of software and not three or four? In this project, the dual development approach is cost-effective for reliability and the other reasons mentioned above. Of course, the use of three or more independently developed programs may lead to better reliability, however, the cost of development will be prohibitively large and hardly justifiable due to a significant increase in complexity.

In the following sections the various features of the methodology will be discussed in more detail.

III. Requirements Specification, Preliminary Design, and Analysis

In this section, we discuss the pilot software, and the experiences in the functional requirements and preliminary design phases of the project.

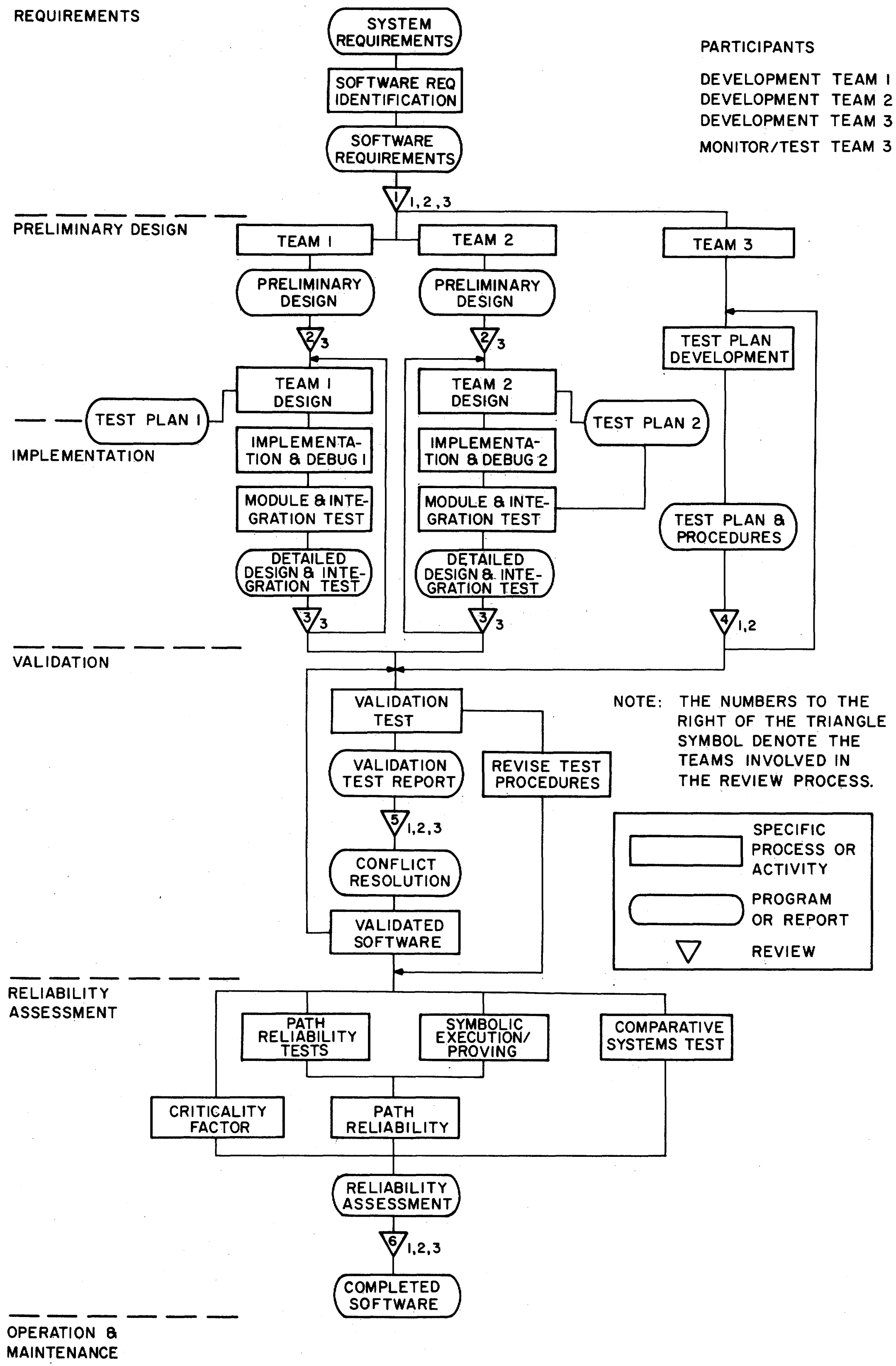


Fig. 1. Dual design methodology.

A. Pilot Software for Project

The operations of the safety protection system can be considered as a set of periodical calculations of the plant status based on reactor signals. If the calculated results indicate abnormal behavior, a signal is given out to trip (shut off) the plant. (Many of the trip functions depend on the power level of the reactor which, however, cannot be measured directly.)

The safety protection system software selected for this project is the "neutron flux signal calibration" function [21]. The function, in essence, tries to estimate the actual reactor power (core thermal power) as closely as possible, based on primary system signals such as the out of core neutron flux signal, inlet and outlet temperatures, pressures, etc.

B. Functional Requirements Specifications

The requirements document for the application, called specification of functional requirement (SFR) [21], is a structured document written in free form English by the nuclear applications specialists. The requirements were formulated in two major sections and accompanied by a set of detailed acceptance test data [22].

Design Bases: Set forth the environment and the physics of the problem.

Specification of Requirements: Represent a specific statement of the requirements and associated algorithms which have been thoroughly validated by the application specialist.

The functional requirements for the application are expressed as 28 separate requirements and 16 more general rules or criteria.

There have been seven sets of major revisions since the first document was released. The reasons for these revisions are as follows:

1) requirements changes such as adding new capabilities, modifying existing requirements, etc.;

2) ambiguity of the requirements, incompleteness of the requirements, and inconsistency among the requirements found during the preliminary phase;

3) incompleteness of the requirements and inconsistency among the requirements found during the detailed design and coding phase.

A list of the functional requirements modifications and the number of their occurrences are given as follows:

- incorrectness (23)
- clarification (21)
- inconsistency (7)
- missing information (5)
- redundant information (15)
- infeasible requirements in a sequential mode (4)
- others (38).

Most of the errors of the functional requirements are detected during the formal specification (preliminary design) phase because of inconsistencies arising during review and comparison of the dual specifications. The large number of revisions to the functional requirements document has emphasized the cost of, and stressed the need for, automating documentation control. Other desirable features include:

1) provision of all needed information but nothing extraneous;

2) if requirements cannot be stated in a closed form, a validated algorithm should be included;

3) all requirements should be separately identifiable and testable;

4) control logic and sequence must be unambiguous.

C. Specification Development and Preliminary Design

As noted in Section II above, two independent teams are engaged to develop the specifications from the functional requirements. An independent third organization coordinates the process and analyzes both specifications (preliminary designs). By this approach, a larger number of errors can be identified at an early stage. To achieve this, a formal specification language is necessary to structure this effort and simplify the evaluations which may be performed by a combination of automated analyses (for certain aspects of consistency and completeness) and by manual reviews. The discrepancies of the dual specifications are reconciled to the mutual satisfaction of the development and review teams. The development teams then carry on the detailed design and implementation of their own specifications.

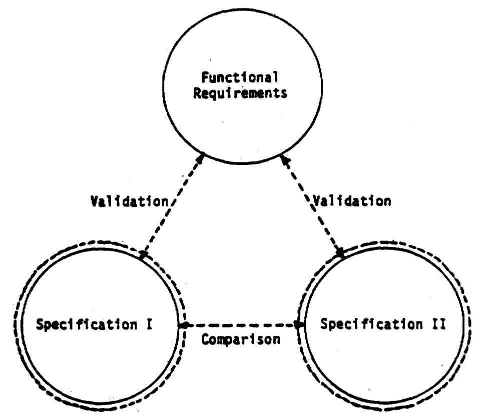


Fig. 2. Dual specification analysis.

There are three aspects of specification analysis, schematically shown in Fig. 2: 1) validation of specifications against the functional requirements, 2) consistency analysis of the specifications, and 3) intercomparison of the dual specifications. Since the functional requirements are the originating document for the specifications, the latter must be verified as to its correct derivation from the former. This is a very difficult task. However, analyzing the specification by itself usually can reveal "glaring" errors and inconsistencies, particularly those syntactically and structurally oriented (although it is less effective for semantic-based errors). Although this can only be considered as partial validation, significant error detection can be achieved, eliminating much later validation effort. The comparison of the two specifications is based on the assumption that independent interpretations of the same set of functional requirements would pinpoint most misinterpretations and ambiguities in the original functional requirements document. Furthermore, as a formal language, requirement specification language (RSL) [1] provided by the Ballistic Missile Defense Advanced Technology Center (BMDATC), is used write the specifications, the dual specifications are more amenable to an automatic procedure for their comparison.

1) Application of RSL for Preliminary Design: A preliminary design (specification) of the application was developed by software engineers of each team. RSL provided a medium for describing the functional elements, data structures, required event sequences, and interfaces. RSL also has a facility of representing structures in graphical forms as well as in textual forms.

In the preliminary design [16], each requirements statement contained in the SFR (specification of functional requirements) was translated into an RSL element called ORIGINATING-REQUIREMENT. Every RSL ORIGINATING-REQUIREMENT traces to one (or more) other RSL elements such as *R*-NET, SUBNET, ALPHA, DATA, INPUT-INTERFACE, OUTPUT-INTERFACE, MESSAGE, etc. [1]. Relationships among these elements, and their attributes not only characterize the requirements but also provide the basis for validating the preliminary design against its functional requirements.

For example, the traceability feature [1] ensures that every RSL element is related to a specific portion of the functional requirements, and that all requirements are decomposed into some RSL elements. The *R*-NET STRUCTURE provided in RSL as an attribute of *R*-NET or SUBNET ensures that the control logic interconnecting functional modules (lower level SUBNET's and ALPHA's) are unambiguous and that all alternatives at each decision point are addressed. RSL is accompanied by a collection of analysis tools called Requirements Engineering Validation System (REVS) [1]. The latter eliminates syntax errors and helps to document the preliminary designs and ensures the completeness and consistency of the designs. As an example, when a specific functional requirement was modified, REVS was used to list all RSL elements tracing to that functional requirement.

From our experience, the RSL preliminary designs are easier for the reviewer to understand and examine. Using the sorting and selective listing capabilities, the tasks of auditing and review were simplified. The resulting preliminary design document was easier to maintain using the capabilities of REVS.

However, the use of a formal preliminary design language and a third team resulted in a substantial commitment of manpower to learn about the systems (the application as well as the RSL/REVS system) and to perform the necessary mechanics of inputing a large amount of design information. It has also become clear that the deeper the involvement the greater the number of errors which are uncovered. For example, during the functional requirements review process, a number of high-level problems were detected and corrected. However, it was not until the second team initiated their preliminary design that a significant number of other problems were identified. Table I summarizes the differences between the two preliminary design documents. Notice that there is a large difference in the number of paths although both designs have the same requirements.

Fig. 3 illustrates the control flow diagram of the two preliminary designs. After careful analysis and refinements the number of paths was reduced in Team 2's design resulting in a more reliable implementation. RSL/REVS is still an experimental tool which needs further improvements before it can be used successfully in a commercial environment. Specifically, computer resource requirements need to be reduced in terms of memory size and execution time. For our application, the memory on the CDC 7600 was frequently overcommitted and the execution time (central processor time) was over 30 s for a single update and a listing of the database.

2) Preliminary Design Experiences: The Software Preliminary Design Review (SPDR) process conducted after completion of the preliminary design uncovered many errors in the design documents. Typical errors included:

1) incomplete data statements,
2) correct, but poor design,
3) untraceable statements to the ORIGINATING-REQUIREMENTS.

These errors were detected 1) by manually comparing the preliminary design document and the functional requirements and 2) by using REVS. Table II summarizes the results of the manual review and Table III summarizes the results of the analysis tools.

TABLE I

Preliminary Design Statistics		
RSL Structure	TEAM1	TEAM2
# R-Nets	3	1
# Sub-Modules (ALPHAs)	24	28
# Interfaces	2	2
# Paths	448	36

The design process also detected several deficiencies in the functional requirements document. These are summarized in Table IV.

In conclusion, the preliminary design phase has resulted in the detection and correction of many significant errors in both the functional requirements and the preliminary designs. These occurred early in the software development cycle when errors are easier and less costly to correct. The dual development approach contributed significantly in the identification of errors and ambiguities in the functional requirements.

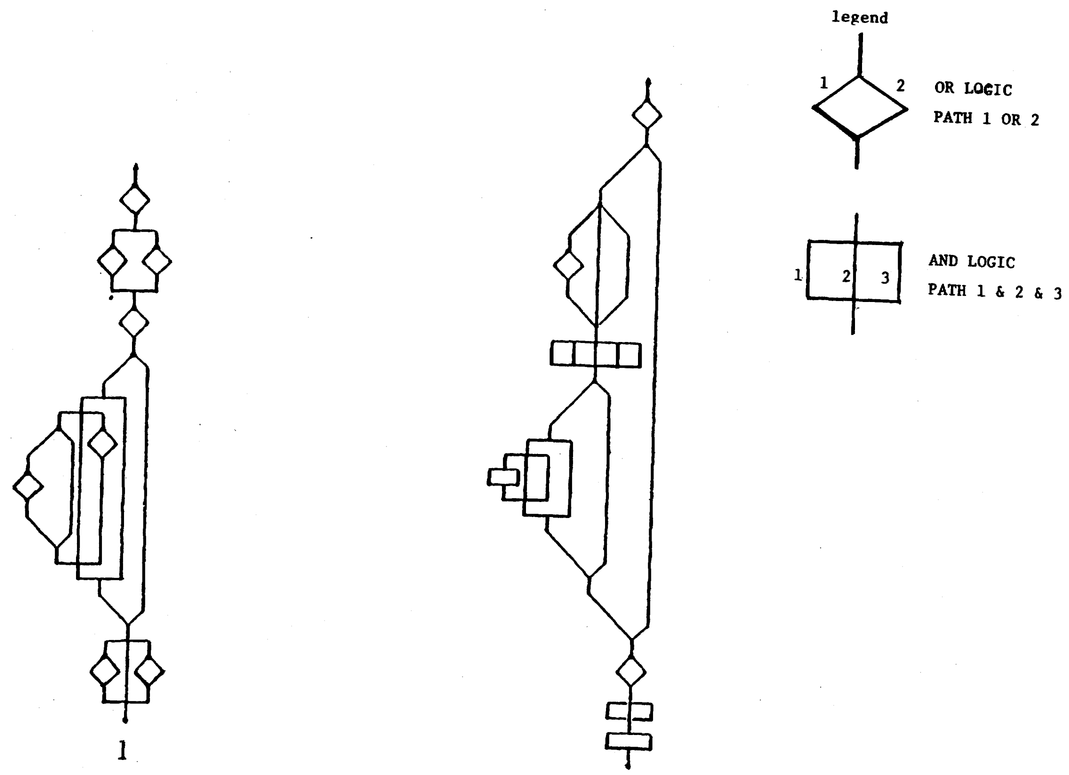
IV. Detail Design and Implementation

The design process is one in which a design is synthesized from the software requirements specification. The resulting design should contain sufficient information for a straightforward implementation. In this project the detail design activity is to incorporate input/output assertions into each of the ALPHA's (modules) as defined in the preliminary design phase in RSL. Fig. 3(b) illustrates the incorporation of I/O assertions for each ALPHA into the RSL description in the detailed design phase.

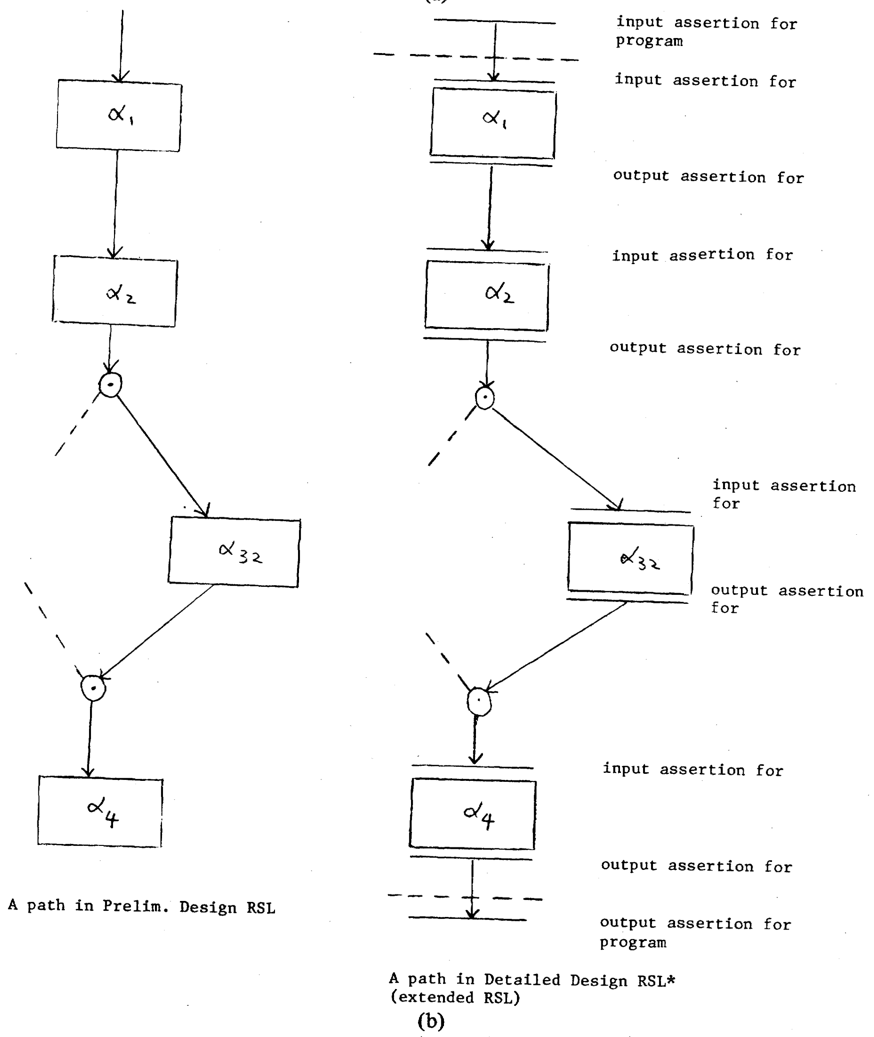
An input/output assertion is a description of the relation between the input and output variables in a module (e.g., ALPHA, subroutine, etc.). Fig. 4(a) shows the input/output assertion for an ALPHA for the application program. The assertions are expressed in a Fortran-like syntax using some additional notations from first-order logic. Unlike the high-level input/output assertions usually written for some routines (for example, a sort routine), we have found it more natural to develop low-level, expression-oriented assertions for the routines in our application program. Also, we have used temporary variables for the more complicated ALPHA's in order to facilitate the specification of the assertions, even though these could have been written without using any temporary variables.

The input/output assertion for each ALPHA is augmented by a list of the input and output variables and the input assertion (constraints on the input variables). This enables the coding to be done with minimal reference to the RSL description of the specifications. (In our project, two versions of the application program were developed in less than a week by two different programmers.)

One problem is that the input/output assertions are extremely difficult to generate from the specifications. What would be helpful is to have some tools for assisting the generation and checking of the resultant I/O assertions. Unfortunately, REVS does not have the analysis tools for checking the validity of the input/output assertions. Thus we were forced to check the assertions manually. The review was performed by the



(a)



(b)

Fig. 3. RSL path enumerations (Team 1 and 2 preliminary designs).

TABLE II

Deficiencies in Preliminary Design Found During Review (Manual)	
Categories	Frequency
Missing data group definitions by referring to other places such as TABLES (non-standard RSL)	5
Insufficient or wrong traceability to originating requirements	12
Uninitialized data values	33
Design Errors (correct interpretation of F.R., wrong design)	1
Improved Design (better design practices: e.g., simplification of formula, handling of spare slots of memory space, direct and clear traceability to originating requirements, sufficient comments, consideration for overflows, etc.)	12
Spelling Errors	2

TABLE III

Deficiencies in Preliminary Design Found During Review (using REVS)	
Categories	Frequency
Data Flow Anomalies	4
Incomplete Design:	
• Data Definition	6
• Validation Points/Validation Path	4
• Performance Requirements	1
• Not traced from Originating Requirements	4

TABLE IV

Functional Requirements Deficiencies Found During Preliminary Design	
Categories	Frequency
Redundant data definitions	2
Inconsistencies in F.R.:	
• Processing order of functions	2
• Data item spelling errors	1
• incomprehensible	2
Ambiguous or incomplete requirements	7

```
ALPHA:  CF1_CALC.
     DESCRIPTION:  'CALCULATE THE CONTROL ROD POSITION FACTOR, CF1.'.
     INPUT_ASSERTION:  '     TRUE'.
     OUTPUT_ASSERTION:
      'CF1 .EQ. ((PRODUCT. I=1,7  (1+AR_I*(FFL(ZRIIX5)+(FFL(ZRIIX5+1)
                 -FFL(ZRIIX5))*(ZRII/5-ZRIIX5))))*
                 (1+AR_8*(FPL(ZR18X5)+FPL(ZR18X5+1)-FPL(ZR18X5)*
                 (ZR18/5-ZR18X5))))'.
     INPUTS:
          DATA:  ZRI_STATE.
     OUTPUTS:
          DATA:  CF1.
     TRACED FROM:
          ORIGINATING_REQUIREMENT:  FR_R6_5_4_8_2.
     REFERRED BY:
          SUBNET:  COMPUTE_PHIN_HAT.
```

(a)

```
      SUBROUTINE BCF1CL(CF,ZR)
C*************************************************************
C  FUNCTIONAL DESCRIPTION
C     CALCULATE THE CONTROL ROD POSITION FACTOR,CF(1)
C  ALPHA DESCRIPTION
C        BCF1CL(CF1-CALC)
C  IDENTIFIER DESCRIPTION
C     CF        VARIABLE  PARAMETER OUTPUT
C     ZR        VARIABLE  PARAMETER INPUT
C     AR        CONSTANT  GLOBAL    INPUT
C     FFL       CONSTANT  GLOBAL    INPUT
C     FPL       CONSTANT  GLOBAL    INPUT
C     I         VARIABLE  LOCAL
C     J         VARIABLE  LOCAL
C     JJ        VARIABLE  LOCAL
C  CALLED BY
C     BCPHNH    *COMPUTE PHINH(PHIN-HAT)
C  MODULES CALLED
C     NONE
C  RESTRICTIONS/EXCEPTIONS
C     NONE
C*************************************************************
C     USE LINEAR INTERPOLATION IN THE TABLE OF F(PL)
C  WHERE THE INDEX INTO THE TABLE IS 5X OF THE CONTROL
C  ROD POSITION IN BANK 8 (ZR[8]) TO FIND F(PL).
C           INDEX = INTEGER OF Z[8]/5.Ø
C           DELTA = ZR[8]/5.Ø - REAL OF INDEX
C           F(PL) = FPL [INDEX] + DELTA * (FPL[INDEX + 1]
C                   - FPL [INDEX]
C     SET
C           CF1   = 1.Ø + AR[8] * F(PL)
C     FOR I = 1 TO 7
C     USE LINEAR INTERPOLATION IN THE TABLE OF F(FL)
C  WHERE THE INDEX INTO THE TABLE IS 5X OF THE CONTROL
C  ROD POSITION IN BANK I (ZR[I]) TO FIND F(FL).
C           INDEX = INTEGER OF ZR[I]/5.Ø
C           DELTA = ZR[I]/5.Ø - REAL OF INDEX
C           F(FL) = FFL[INDEX] + DELTA * (FFL[INDEX + 1]
C                   - FFL[INDEX]
C     SET
C           CF1   = CF1 * (1.Ø + AR[I] * F(FL)
C     END FOR
C*************************************************************
C     CONSTANT DECLARATION
      COMMON /BFARCT/AR,FFL,FPL
      REAL FFL(22),FPL(22),AR(8)
C     VARIABLE DECLARATION
      REAL CF(4),ZR(8)
      INTEGER I,J,JJ
C     MODULE PROGRAM CODE
      CF(1)=1.Ø
      DO(I=1,7)
      J=ZR(I)/5
      CF(1)=CF(1)*(FFL(J+1)+(FFL(J+2)-FFL(J+1))*(ZR(I)/5-J)))
      END DO
      JJ=ZR(8)/5
      CF(1)=CF(1)*(1.Ø+AR(8)*(FPL(JJ+1)+(FPL(JJ+2)-FPL(JJ+1))*
     1    (ZR(8)/5-JJ)))
      RETURN
      END
```

(b)

Fig. 4. (a) An ALPHA in RSL. (b) Subroutine corresponding to the ALPHA of (a).

designer and two other persons. In spite of this, 20 errors were found in the input/output assertions during the implementation stage (see below). Later in this project, we have developed an assertion checker [24] which checks among other things for syntax errors in the assertions. It consists of approximately 2500 lines Pascal code, 2000 lines of which are for checking the assertions and the rest for preprocessing the assertions into the standard form. While using the assertion checker, about 70 percent of the errors detected were syntax errors, 15 percent were errors like improper declarations and type information missing, 15 percent were improper type usage. Most of these were typographical in nature, such as spelling errors and mismatched parentheses. However, some of these errors lead to serious misinterpretation of the design, which were rectified later.

A. Experiences

The implementation stage consists of the activities which transform the design into a functional program (B1) in some programming language. Iftran [28], a structured Fortran language, was chosen as the implementation language because of its widespread industrial use. Also, several analysis tools (like, PET [26], FACES [18], etc.) are available for analyzing Fortran programs.

Using the same input/output assertions, another program (B2) was developed to aid in the program testing phase (see Section V). This B2 program, coded in Fortran by another programmer, served as a verification aid for the existing Iftran program.

As mentioned before, the ALPHA's constituting the design are primarily described by input/output assertions. These assertions are so detailed that the implementation is very straightforward. However, naming conventions and data structures which conflict with the syntax of Iftran have to be changed. Because of the Fortran-like syntax of the assertions, coding the B2 program was relatively easy. The pertinent features of the implementation are as follows.

1) In program B1:

- each subroutine in the implementation corresponds to an ALPHA;
- the application program consists of about 1900 lines of structured code;
- each subroutine contains pseudocodes and identifier descriptions to enhance understanding of the subroutine.

As an example, the subroutine corresponding to the ALPHA of Fig. 4(a) is shown in Fig. 4(b).

2) In program B2:

- each subroutine in the implementation corresponds to an ALPHA;
- the application program consists of about 1300 lines of structured code;
- it is a "bare" program, without much documentation since its function is exclusively to aid in testing program B1.

The implementors experienced several difficulties in the coding of the programs. They came across cases of inconsistencies and incompleteness in the input/output assertions. Even though RSL and the input/output assertions are formal descriptions, there were a few cases of ambiguities. Some of these difficulties were traced to defects in the requirements specification and design. Both of these had been thoroughly reviewed. One reason is that personnel involved in developing the software (from the preliminary design to the implementation) had no background in nuclear engineering. Thus requirements that may be obviously infeasible to a nuclear engineer may not be caught until a very late stage when the program has to be actually implemented.

The following sections discuss the testing and validation issues.

V. Validation and Testing

In our methodology for developing reliable software, the validation and testing phases are extremely critical for assuring the quality of the software developed. Our methodology calls for thorough validation procedures to be followed at the completion of each phase in the development process. The various review and validation activities followed will be mentioned when our experiences in the various phases are discussed. In the following subsections, validation procedures employing symbolic execution and the automatic testing tools are discussed. Table V illustrates the tools used during the development phase.

The notion of symbolically executing a program follows quite naturally from normal program execution. Instead of using real data objects, all inputs to a program are assigned symbolic data objects. An instantaneous program state is maintained during the symbolic execution of a statement in a path. The program state consists of the symbolic values of the program variables and the path constraint (PC). The symbolic execution of a path is defined as transformation over the system vector $(A_1, A_2, \cdots, A_m, pc)$ where A_i represents the symbolic value of v_i (the ith system variable) and pc represents the path condition which is a predicate describing the symbolic inputs to the program. Initially each A_i is undefined and $pc = T$. In executing an assignment statement symbolically, the arithmetic expression is evaluated following the rules of algebraic manipulations and the symbolic result is used to update the corresponding output variable in the program state. Path constraints are updated during the execution of conditional statements. For example, if the statement IF <*booleanexpression*> THEN <*statement*$_1$> ELSE <*statement*$_2$> is executed, and the true branch is taken, then PC $\leftarrow$ PC $\wedge$ <*booleanexpression*>. The path constraint is a set of equalities and inequalities describing the program inputs such that input data satisfying these constraints will lead to the execution of that statement. Program validation using symbolic execution has been found useful for 1) deriving the outputs as a function of inputs for manual inspection, 2) generating test cases automatically, and 3) deriving verification conditions for correctness proofs. In the following subsections, we will discuss some of the uses of symbolic execution for validation.

1) High-Level Symbolic Execution of Specifications for Intercomparison: The use of a formally defined language such as RSL is an initial and important step towards an approach for the comparison of specifications. A major feature of RSL is its ability to model the system response to stimulus as an *R*-net. Comparison of dual specifications can be based on symbolic execution of the *R*-nets which closely resemble that of a program. It has long been recognized that the size of the symbolic expressions is the limiting factor in the symbolic execution of programs. Applying symbolic execution at an earlier phase where programming details are not present will alleviate this problem. An *R*-net with parallel tasks is first converted into an equivalent *R*-net without parallel tasks by sequentializing them in an arbitrary order. Paths or execution sequences are then identified from the *R*-net. Each path at the specification level consists of a sequence of ALPHA's whose functional characteristics are defined by its corresponding input and output assertions. During *high-level symbolic execution*, the function of each ALPHA is represented by the I/O assertion for the ALPHA in question (Fig. 3(b)). In traversing the path during high-level symbolic execution the transformation of the path constraints and system variables is solely a function of the I/O assertion of the alphas along the path. This is performed using the I/O Assertion Checker [24]. The internal details of the ALPHA is not considered at this level. Validation of the individual ALPHA's with respect to the I/O assertions is accomplished using symbolic execution at the Fortran level as described in the next section. For each RSL constructs, transformation to the system state vector $(A_1, A_2, \cdots, A_m, pc)$ during execution is defined. When symbolic execution is completed, the path condition gives the conditions on the inputs that lead to the execution of that path, and A_i gives the symbolic expression of v_i upon exit.

When the two specifications are compared, one of them, say A, is used as the standard against which the other, say B, is compared. A path in A is identified and symbolic execution is used to derive its path condition and output expressions. Using this path condition, corresponding path(s) in the other

TABLE V
SUMMARY OF SOFTWARE TOOLS

Development Phase \ Tools	Requirement Statement Language	Requirement Engineering and Validation System (REVS)	Static Analyzer	Dynamic Analyzer	Dual Testing Tools	Symbolic Executor	Test Case Generator	Theorem Prover	Input-Outpu Assertion Checker
Specification	✓								✓
Spec. Analysis		✓							
Design Analysis		✓							
Implementation			✓						✓
Module Testing				✓		✓	✓		✓
Integration Testing				✓	✓	✓	✓		✓
Reliability Assessment 1) criticality factor				✓					
2) correctness factor						✓		✓	
3) machine factor									
4) compilation factor			✓						
Development Effort in Man-months			24	12	6	15	12		3
Size (lines of code)		40,000	8,000	3,000	1,500	6,000	5,000		2500

specification is identified and output expressions are derived. These two sets of output expressions and path conditions are finally compared and the discrepancies analyzed. Fig. 5 illustrates how the capabilities of A are mapped into capabilities of B using the above procedure. Area y corresponds to capabilities common to both specifications A and B. Area x corresponds to capabilities which exist in A and not in B. Note that capabilities that are provided by B, but not A (area z) cannot be detected by this procedure. However, by reversing the role of specifications A and B, i.e., use B as the standard to check A, the last class of discrepancy can be detected. Since this procedure is carried out at a higher level (specifications rather than code), powerful heuristics for simplifying expressions and manipulating input and output assertions will be needed. Thus, it is extremely difficult to mechanize and this procedure is mainly manual. The procedure to resolve discrepancies is expected to be fairly complicated since not being identical does not necessarily mean that one specification is wrong and other is correct. Individualized analyses are expected and a certain extent of judgment is definitely involved.

2) Symbolic Execution for Program Verification: Symbolic execution is also used to generate verification conditions mechanically for proving the correctness of an execution path. In Fig. 6, let $I(\bar{x})$ and $O(\bar{y})$ be the input and output assertions and let α be an execution path from program entry to exit; then to prove that α is correct, we have to show that

$$I(\bar{x}) \wedge PC_\alpha(\bar{x}) \Rightarrow O(R_\alpha(\bar{x}))$$

where $PC_\alpha(\bar{x})$ are the path constraints and $R_\alpha(\bar{x})$ gives the value of the program variables in terms of the program inputs and the initial values of variables.

To prove that a program is correct, we have to show that every path in the program is correct, i.e.,

$$\bigwedge_{i \in \varphi} [I(\bar{x}) \wedge PC_i(\bar{x}) \Rightarrow O(R_i(\bar{x}))]$$

where φ is the set of execution paths in the program. For programs with a large number of paths, this may not be a feasible approach to prove correctness. However, for the critical applications we are considering, the number of paths is usually small.

3) The Symbolic Execution System: The design and implementation of the symbolic execution system for this project serves as an interesting software engineering study in the incorporation of large sophisticated special purpose systems into a general purpose system for program manipulation. The Fortran-to-Lisp project at M.I.T. [17] provides our system with a production quality input language processor. The mathematical typesetting facility residing on the Berkeley VAX/UNIX [8] provides sophisticated output of symbolic mathematical formulas. Finally, the MACSYMA symbolic and algebraic manipulation system [12] forms the backbone of the formula manipulation capability of our system. All of these systems are linked together through the Berkeley INGRES 11/70 ARPANET host.

The construction of the symbolic executor over MACSYMA

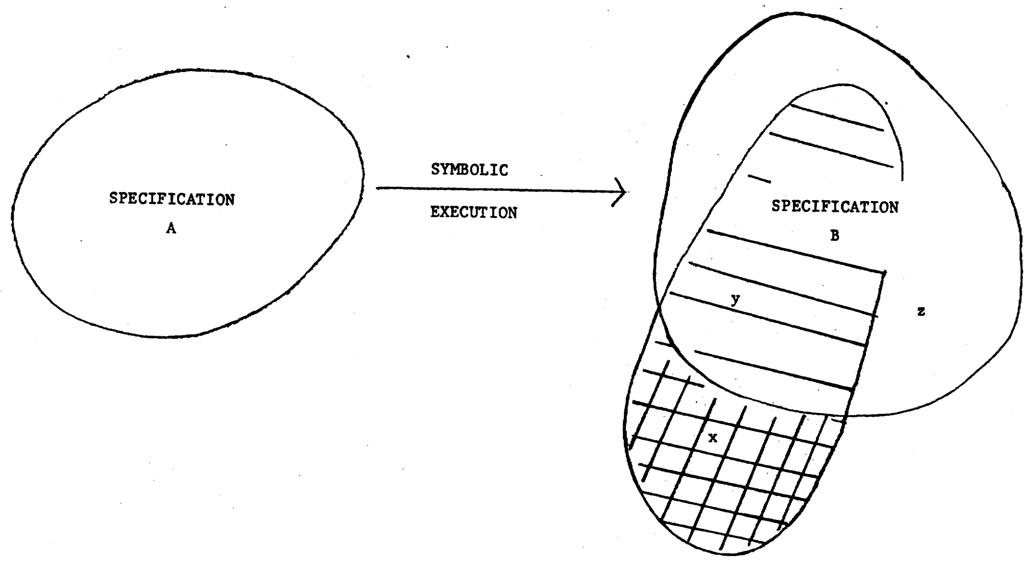


Fig. 5. Specification analysis.

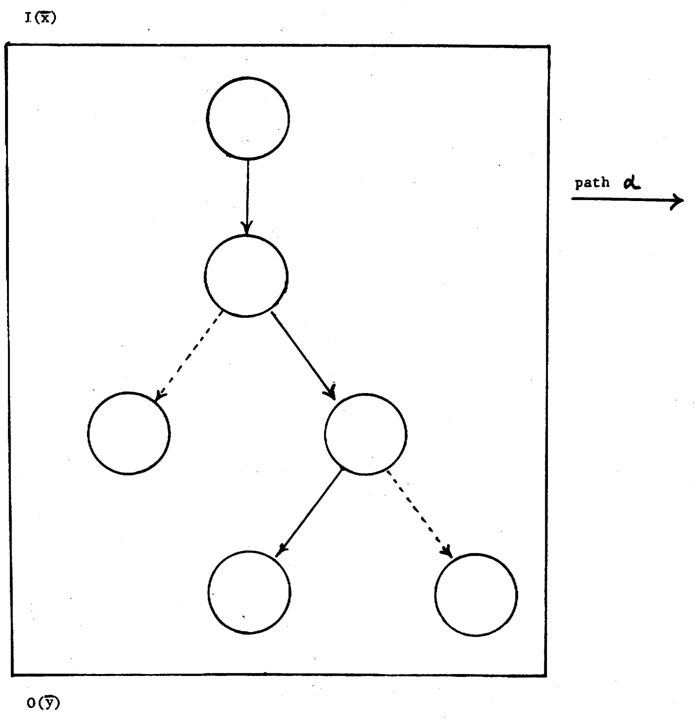


Fig. 6. Path analysis.

reflects our belief that the proper role model for symbolic execution is the system for symbolic mathematics, rather than the general purpose programming language execution system. Although other systems for symbolic mathematics are available, such as REDUCE, the MACSYMA system is certainly the most ambitious of these.

The presence of a system for symbolic mathematics is especially important to the application area we are dealing with in our project, namely process control software. Such software involves heavy computation, placing equally heavy demands on the formula manipulation capabilities of the symbolic executor.

Among the facilities made available through MACSYMA that have proven to be most valuable to us have been: a powerful mechanism for the two dimensional display of expressions; the general purpose simplifier; user-controlled pattern-matching; and a flexible conversational environment.

The nature and operation of the system is illustrated by the symbolic execution of a typical module from the safety system software. Fig. 7(a) shows a typical subroutine, together with a small calling program. They were written in Iftran, and are shown here after translation into Fortran IV, which is accepted by the system. The input language has been augmented by nonstandard statements pertinent to symbolic execution. For instance, note the SYMREAD statements of the calling program whose purpose is to assign symbolic values to their arguments. They are prefixed by special comment sequences allowing them to be introduced directly into the code. Symbolic values assigned to variables take the form of lowercase names.

Fig. 7(b) is a protocol of the symbolic execution of this program. Annotations to the protocol appear enclosed in asterisks. In this program there are two possible branch points, both involving the value of the variable *CC*. An interactive resolution of these branches takes place during the course of execution. The MACSYMA conversational mode consists of labeled lines which can be later referenced by the user.

In addition to the user-controlled mode of conditional resolution shown here, there are also automatic and path-directed modes of resolution available.

After completion of symbolic execution, all symbolic values reside the MACSYMA workspace, and the user is free to interact with MACSYMA to inspect, manipulate and store symbolic values. The protocol shows the inspection of three typical values, illustrating the two-dimensional display at the same time. In this particular program we find a mixture of numeric and symbolic calculations, as reflected in the output values. MACSYMA has the capability for arbitrary-precision floating-point arithmetic.

The last value to be inspected in the protocol is the *PC*, or Yet it is a matter of judgment to select the most appropriate form: in this unsimplified state, the *PC* yields a clear trace of the path through the program which was taken. However, a more simplified form may be desirable when validating the program.

A more detailed discussion of the symbolic executor may be found in [7].

4) Symbolic Execution Experiences: Symbolic execution is used in the validation process to generate output expressions which can be visually examined and compared to requirements and specifications. Formal proving is also attempted using symbolic execution and the input/output assertions.

In our project, symbolic execution is performed on a module basis in a bottom-up fashion. This is necessary in order to limit the symbolic execution activities to a reasonable number of path and to contain the growth of the symbolic execution tree to a manageable size for the system. Since we have developed input/output assertions for each module, the symbolic expressions generated can be verified against the I/O assertions and correctness of the module proven. After this is performed, symbolic execution of a higher level module can be performed by substituting calls to lower level modules by the I/O assertions.

Our experiences with symbolic execution indicates that visual inspection of output expressions is very useful for indicating problem areas in the code. With this type of process control software, since output depends on previous cycles of the input, output expressions usually follows a very regular form as a function of the input values in previous time instances; any deviations from the regularity in the expressions are very prominent and usually indicate errors.

```
      PROGRAM BCTPRG
      INTEGER CC
      REAL LAMDC(6),CTPØP,CTPHAT,CTPØHT
      REAL CTPP,EKØ,EK(2),PHINH
C&    SYMREAD CC,LAMDC,CTPØP,CTPHAT,CTPØHT
C&    SYMREAD CTPP,EKØ,EK,PHINH
      CALL BCTPØP(CC,LAMDC,CTPØP,CTPHAT,
     1 CTPØHT,CTPP,EKØ,EK,PHINH)
      STOP
      END
C
C
C
      SUBROUTINE BCTPØP(CC,LAMDC,CTPØP,CTPHAT,CTPØHT,CTPP,
     1 EKØ,EK,PHINH)
      INTEGER CC
      REAL APHNKT,LAMDC(6),SLMCKT,SPHNKT
      REAL CTPØP,CTPØHT,CTPHAT,EKØ,EK(2),PHINH,CTPP
      DATA SPHNKT,SLMCKT,APHNKT/Ø.Ø,Ø.Ø,Ø.Ø/
      LAMDC(1)=Ø.915303*LAMDC(1)+Ø.ØØØ254*PHINH
      LAMDC(2)=Ø.971562*LAMDC(2)+Ø.ØØØ236*PHINH
      LAMDC(3)=Ø.996636*LAMDC(3)+Ø.ØØØØ521*PHINH
      IF(CC.EQ.Ø) GO TO 19997
      GO TO 19998
19997 CONTINUE
      SPHNKT=PHINH
      SLMCKT=(LAMDC(1)+LAMDC(2)+LAMDC(3))
      GO TO 19999
19998 CONTINUE
      SPHNKT=SPHNKT+PHINH
      SLMCKT=SLMCKT+LAMDC(1)+LAMDC(2)+LAMDC(3)
      IF(CC.EQ.39) GO TO 19994
      GO TO 19995
19994 CONTINUE
      APHNKT=SPHNKT/4Ø.Ø
      LAMDC(4)=Ø.987549*LAMDC(4)+Ø.ØØØ239*APHNKT
      LAMDC(5)=Ø.999Ø53*LAMDC(5)+Ø.ØØØØ11*APHNKT
      LAMDC(6)=Ø.9999Ø4*LAMDC(6)+6.25E-7*APHNKT
      EK(2)=EK(1)
      EK(1)=EKØ
      EKØ=Ø.935738*APHNKT+LAMDC(4)+LAMDC(5)+LAMDC(6)+SLMCKT/4Ø.Ø
      CTPP=Ø.6Ø6531*CTPHAT+Ø.393469*EK(2)
      CTPØP=Ø.6Ø6531*CTPØHT+Ø.393469*CTPP
19995 CONTINUE
19996 CONTINUE
19999 CONTINUE
      RETURN
      END
```

(a)

Fig. 7. (a) Typical subroutine with calling program for symbolic execution.

A. Test Data Generation and Dual Program Comparison

Testing remains an indispensable tool for demonstrating the correct implementation of a program. Early planning is of paramount importance if testing is to be carried out effectively. In our methodology, test planning forms an integral part of the specification, design, and implementation process. Considerable attention was given to choosing test cases during the various stages of the development process so as to be effective and representative. If one chooses suitable test cases, one could have a fair degree of confidence in the correctness of a program.

For the pilot software, acceptance test cases were obtained from simulation data of the various simulation models used by the nuclear reactor vendor, Babcock and Wilcox [22]. These test cases require approximately 24 hours of machine time. They will provide confidence that the system performs correctly under most operating conditions.

A test plan and test cases based on the functional requirements have been developed by the independent team and the two development teams. These test cases cover all the capabilities described by the requirements and are independent of the specifications and preliminary design. During the specification process, additional test cases were derived based upon the *R*-NET's. Each path in an *R*-NET specifies the system response to stimulus [1] and test cases were generated to exercise all stimulus-response in the *R*-NET. Additional test cases were identified to test the implementation. These include test cases derived from the input/output assertions of the software system as well as those of the individual modules. Some test cases were derived based on the algorithm used in the specific application. These include boundary and interior values (including out of range values) to validate the functional form the various expressions used in the algorithm. Finally, integration test cases are identified to test interfaces among modules and some internal logic.

Thus, test cases are chosen during each phase of the software development, forming a hierarchy of test cases. For example, test cases derived from the requirements specification test for broad characteristics while those based on the implemented code test for finer details. The former test cases check for gross (large "size") errors while the latter check for subtle (small "size") errors. The advantages of such a hierarchical set of test cases are 1) faster verification of the code after a modification, 2) systematic location of errors, and 3) it is simpler to derive a complete set of test cases from the requirements specification than from the code (which can contain a very large number of paths). The test cases are exercised in the hierarchical order.

The application program in this project maintains a history of past inputs as internal states. The outputs are a function of

```
(C95) EXECUTE(BCTPRG)$                    **** Invoke symbolic execusion ****
                                          **** of BCTPRG program         ****
----------------------

[BCTPOP, 9]     IF(CC.EQ.0) GO TO 19997
UNRESOLVED CONDITIONAL:
1  :  EQUAL (cc, 0)                          **** Symbolic values in     ****
2  :  cc # 0                                 **** conditional predicate ****
SELECT RELATION NUMBER:
2;
----------------------

[BCTPOP, 9]
ASSUMED:  cc # 0                             **** Resolved interactively ****
----------------------

[BCTPOP, 18]     IF(CC.EQ.39] GO TO 19994
UNRESOLVED CONDITIONAL:
1  :  EQUAL(cc, 39)
2  :  cc # 39
SELECT RELATION NUMBER:
1;
----------------------

[BCTPOP, 18]
ASSUMED:  EQUAL(cc, 39)       ****Resolution of second branch****
----------------------

[BCTPRG, 7]    END OF SYMBOLIC EXECUTION    **** Return to MACSYMA ****

(C96) EKO;     ****Interrogate workspace for symbolic values ****
               ****of some typical variables

(D96) 0.0233997155 (sphnkt + phinh) + 0.025
 (slmckt + 5.421E-4 phinh + 0.996636 lamdc3 + 0.971562 lamdc2

+0.915303 lamdc1) + 0.999904 lamdc6 + 0.999053 lamdc5 + 0.987549 lamdc4

(C97) CTPP;

(D97)                          0.6Ø6531 ctphat + 0.393469 ek1


(C98) LAMDC[4];

(D98)             5.975E-6 (sphnkt + phinh) + 0.987549 lamdc4

(C99)  PC;      **** Print out value of Path Condition ****

(D99)                 cc # 0   AND  EQUAL(cc, 39)
```

(b)

Fig. 7. (*Continued.*) (b) Protocol of the symbolic execution of module in (a).

the inputs and the states. Unless the state variables can be accessed and set by the tester, it is difficult to force the program to exercise a particular function. In addition, consecutive inputs to the program are related to each other and represent some physical phenomenon, e.g., temperature variations. Hence, the concept of input trajectory is useful. A trajectory is a sequence of test inputs which may be used to bring a program into a particular internal state for testing or just a sequence of related inputs. The use of input trajectories as test cases eliminates tests which are physically unrealizable and reduces the input space considerably. In this application, trajectory testing is used extensively.

Some examples of representative test cases developed for the pilot software are given in [5], [13], [22].

1) Dual Program Testing: The methodology calls for the independent development of dual sets of specification and, from these, two sets of application software. This dual programming technique plays a dominant role in testing. The dual set of software developed can be easily checked by running the same set of inputs on both programs and comparing the corresponding outputs. Since it is unlikely that both programs contain the same errors (specifications and designs are developed independently), discrepancies among outputs (outside certain tolerance limits) may indicate problem areas. With conventional testing approaches, a detailed manual output analysis is required for checking the validity of the outputs. Thus dual programming greatly reduces the manual effort required for the validation of test results and allows a large number of tests to be performed and checked in relatively short time.

The efficiency of dual program testing is further enhanced by tools which aid in the generation of test data and automatic comparison of output values.

2) Dual Program Monitor and Test Data Generator: An automated test data generator and a dual program monitor have been developed to accompany the methodology. Inputs that are generated by the test data generator are applied and outputs are observed under the control of the dual program monitor. Discrepancies in functions and performance are noted for review.

In Fig. 8 the dual program monitor is shown to interact with the two programs and the test data generator and some common database. The major objectives of the dual program monitor are 1) to monitor the execution, 2) to compare the results,

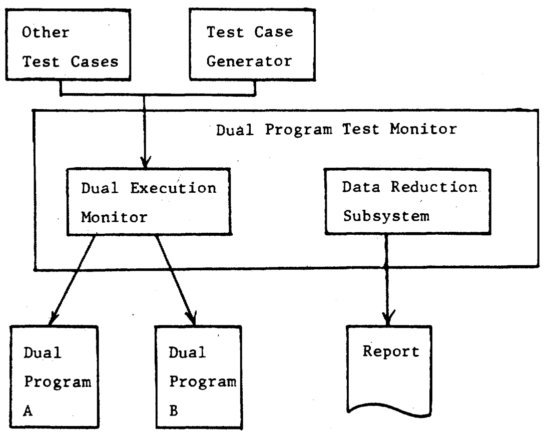


Fig. 8. Dual program monitor and test data generator.

and 3) to collect statistics about the dual programs without modifying the code of either program and maintaining complete isolation between the programs during execution. Since the software under test has to be the same as the software used in the protection system, the monitoring function is performed at the end of each cycle and all relevant information is made available through the interface buffers. Complete isolation between the two sets of software is achieved by having distinct function names of the programs and providing each program with a private copy of all input buffers.

The test data generator [3] is capable of generating test data in the following ways.

1) It generates test cases that satisfy some specified logical relationship among variables (e.g., path conditions obtained by symbolic execution).

2) It generates random test cases according to a distribution over the program input domain.

3) It generates input trajectories (i.e., a sequence of related inputs) given the correlation among variables, rate of change, etc.

Aside from generating inputs as described above, the test data generator is capable of directing inputs from input files (such as actual recorded data or generated from simulation models) to the dual program monitor.

The test data generator and dual program monitor have been implemented and are currently running on a PRIME 400 computer with virtual memory support. The PRIME version of the test data generator and dual program monitor consists of approximately 5000 Fortran statements and is completely interactive. Their design and implementation took approximately 9 man-months for completion. The test data generator takes approximately 20 to 30 ms to generate one test case (for 29 input variables). The dual program monitor takes approximately 50 ms to execute the dual programs for one cycle. On the average, the tools can perform 11 tests/s. The major bottleneck in the testing process is the I/O time required for recording test cases and results in test-log files. The testing process has taken approximately 650 h of machine time. Experiences in using the test data generator and dual program monitor for dual testing will be presented in the next section.

B. Experiences in Program Testing

1) Testing Procedure: The two programs (B1 and B2) were developed according to the same input/output assertions.

TABLE VI

Error Statistics from Dual Testing		
ERROR	B1	B2
Syntax	47	18
Wrong Identifier Name	4	8
Due to Design Changes	4	5
Bad Expression	0	14
Missing Statement/Control Flow	5	7
Missing Declaration/COMMON/DATA	2	2
Bad Dimensioning	2	0
Bad Subroutine Parameter	1	0
Unused Variables	5	0

After several independent compilations of the programs, the syntactical errors were removed and the two programs were tested using the test data generator and dual program monitor.

During the dual testing phase, the testers were able to observe the values of designated variables in the TDG test buffer for a specified test case. Inconsistencies in the values of a variable signifies that either one or both of the programs contain errors. The error location process is a long and tedious process. However, provision is made for the tester to observe values of internal variables in order to track down the cause of the errors. The testing proceeds by exercising one test case at a time and subsequent test cases are exercised only upon successful completion of the previous test cases.

2) Error Statistics: The errors detected during the testing process were analyzed and classified as shown in Table VI.

3) Explanation of the Errors: It is somewhat difficult to provide an adequate explanation as to the cause of all the errors encountered during the testing phase. Many of the errors, especially the design errors were due to the many revisions in the functional requirement specifications (see Section III-B). A few of these changes may have been overlooked and not incorporated in the I/O assertions. Upon completion of the detailed design, it was sometimes necessary to go back and incorporate additional changes in the functional requirements. Because of the differences in the level of complexity and intricacies between the requirements specification and the implementation, there was sometimes a need to change the former to ease coding and readability.

Other design errors were caused by unbalanced parentheses in the input/output assertions. In some ALPHA's, parentheses were nested to five levels or more which caused difficulty during the coding phase. Typographical errors and inconsistent variable names in the input/output assertions also posed problems for the implementor. Some typographical errors were relatively easy to detect. However, those which were buried in a block of arithmetic code were harder to locate. Some variables that had names which were similar in spelling (for example, FFL and FPL), were more difficult to locate than others.

Problems due to errors in the I/O assertions were the most difficult to locate. The error had to be traced back several levels to the functional requirements in order to correct the error. Fortunately, the frequency of such errors were minimal.

It should be noted that there were no analysis tools for checking the validity of the input/output assertions and the programs were coded according to the original version of the I/O assertions. Since then, we have developed an assertion checker [24] which checks for consistencies of the assertions; however, validation of assertions against requirements is still impossible to be mechanized.

After a critical analysis of the error data the following question was raised: which of these could have been prevented by better language features, methodology?

There were some errors that were difficult to locate due to some restrictions imposed by the design specifications and the programming language. The following are examples of identifier names that were incorrectly used:

1) ACKMD for ACKDMD
2) RTEST for BRTEST
3) TALT for TA1LT
4) CC for CL
5) TCSA1 for TRCSA1

There may have been a decrease of such errors if the number of letters for the variables were not restricted. For example, the identifiers OPMP1, OPMP2, OPMP3, can be confusing, whereas SELECT_MANUAL, SELECT_AUTO, SELECT_TRIP are more mnemonic. Errors such as these are easily detected. However, they may take time to locate.

Other errors were found in indexing array dimensions and in missing declarations in COMMON and DATA statements. These are potential areas for improvement.

For example, in the pseudoread routines arrays (A) were used where lists (B) are more appropriate:

```
(A) OPMP1 = LINP(1); OPMP2 = LINP(2);
    OPMP3 = LINP(3); OPMP4 = LINP(4);
    OPMP5 = LINP(5); OPMP6 = LINP(6);
    HCMP1 = LINP(7).
(B) [OPMP1,OPMP2,OPMP3,OPMP4,OPMP5,OPMP6,
    HCMP1] := LINP(1 to 7)
```

Clearly, the code in (B) is more "compact" resulting in larger error sizes since errors in "LINP" or its index are more easily detected. Also, it is easier to modify since additional variables in the array list on the left are implicitly assigned to a variable-index combination by position.

The frequency of missing declaration of COMMON and DATA statements could be attributed to the nature of such statements. A suggestion is in using structures similar to Algol/Pascal's global variables and modules (class structures) employing statements like

```
read<module name>.<identifier name>
write <module name>.<identifier name>.
```

All the above concerns with issues in the design of programming language constructs. Increasing the "size" of errors (if any) of the individual constructs will greatly facilitate the task of error detection and location and simplify the testing process.

VI. Reliability Assessment

A necessary step in the licensing process of nuclear reactors is the assessment of the reliability of the control system. The reliability of the hardware can be estimated by methods based on sound theoretical principles [2]. However, to date no wholly satisfactory theory has been developed for estimating the reliability (or correctness) of the software. The Nelson model [27] is theoretically valid, but it requires a large amount of testing in order to achieve a high confidence bound on the reliability estimate. Also, it assumes random sampling of the input space. This may not be true in cases where the inputs are correlated in time. In this section we will briefly describe the problem of reliability assessment as applied to the project.

Software reliability has been defined as the probability that a *software fault* which causes *deviation* from required output by more than *specified tolerances* in a *specified environment*, does not occur during a *specified exposure period* [27]. Thus, the software need be correct only for inputs for which it is designed ("specified environment"). Also, if the output is correct within the specified tolerances in spite of some error, then the error is ignored. This may happen in the evaluation of complicated floating point expressions where many approximations are used (e.g., polynomial approximations for Cosine, Sine, etc.).

Several software reliability models have been proposed. Some of the models predict the reliability based on the error history [23]. These models treat the software as a black box. Others estimate the reliability based on the results of tests after the debugging phase [23], [6]. Errors found during the reliability estimation phase are not corrected. These models take into account the program structure.

Below we discuss a reliability growth model used during the debugging phase. We also discuss an experimental approach, based on error seeding, in assessing how thoroughly the software has been tested.

Let

$$\alpha_i = P\{\text{success on a run} \mid i \text{ errors detected and corrected}\}$$

where a "run" denotes the execution of an input. Assuming that no new errors are introduced when an error is corrected, we have

$$\alpha_i = \alpha_{i-1} + \Delta_i.$$

In order to model the case where the correction of an earlier error increases the reliability more than the correction of later errors (in a stochastic sense), we have

$$\Delta_i \underset{st}{\leqslant} \Delta_{i-1}.$$

Obviously, $\Delta_i \in (0, 1 - \alpha_{i-1})$. It is assumed to have the distribution $(1 - \alpha_{i-1})^* X$, where X has the beta (r, s) distribution $(r > 1; s > 1)$. Assuming that initially there is an error present for any input (i.e., $\alpha_0 = 0$), this yields [19]

$$E[\alpha_i] = 1 - a^i, \qquad \text{where } a = \frac{s}{r+s}$$

$$\text{MTTF}_i = \left(\frac{r+s-1}{s-1}\right)^i \simeq \frac{1}{a^i}, \qquad \text{for } s \gg 1$$

where $E[\cdot]$ denotes the expectation and MTTF_i is the mean time (number of "runs") to failure after i errors have been detected and corrected.

Thus, on the average this model is equivalent to the Jelinski-Moranda geometric deeutrophication model [14]. Assuming the average behavior, the maximum likelihood estimate of a is obtained by solving

TABLE VII

i = error no.	$\frac{1}{\hat{a}}$	$\widehat{MTTF} = [1/\hat{a}]^i$	actual MTTF
0	-	-	1
1	2.0	2.0	1
2	1.59	2.5281	2
3	1.508	3.43338	10
4	1.676	7.89035	6
5	1.606	10.6838	1
6	1.5015	11.4591	198
7	1.8436	72.3883	203
8	1.874651	152.509	3598
9	2.16678	1052.77	49
10	2.115301	1793.58	1611
11	2.090145	3326.14	9986
12	2.105023	7569.93	-

$$\sum_{j=0}^{i} n_j j \hat{a}^j = \frac{i(i+1)}{2}$$

where n_j = number of successful runs between the $(j-1)$th and the jth failure.

We now discuss the application of this model to the error data derived from the OCED Halden reactor project [4] as well as our pilot software project (the EPRI project).

Like the EPRI project, the Halden project involved research on the development methodology for critical software for nuclear power plant safety control systems. A major problem is the validation of the software and the assessment of its reliability. Table VII and Fig. 9 show the application of the reliability model to the Halden project data. Table VII shows the estimate of the constant "a" and the predicted MTTF's. From a comparison of the predicted and actual MTTF we cannot conclude much regarding the validity of the model. However, we make two important observations. Firstly, the estimate of "a" shows rapid convergence, so that by the jack-knife technique of Mosteller and Tukey [15] we can conclude that the fit is reasonable good, i.e., additional data do not change the model parameters much. Secondly, from Fig. 9 we see that the error data lie largely within the 90 percent upper and lower confidence bounds. The fit is relatively good considering the large fluctuations in the actual data.

In the EPRI project the two programs B1 and B2 (see Section IV-A) were tested with the same set of test cases, as discussed in Section V-B. Since the testing was not random, we have to modify the above derivations.

We consider the mixed function testing process, i.e., a function is tested a certain number of times, then a transition is made to another function (which may have been already tested), and so on. The MLE of "a" is determined by solving the following equation for $\hat{a}$ [19]:

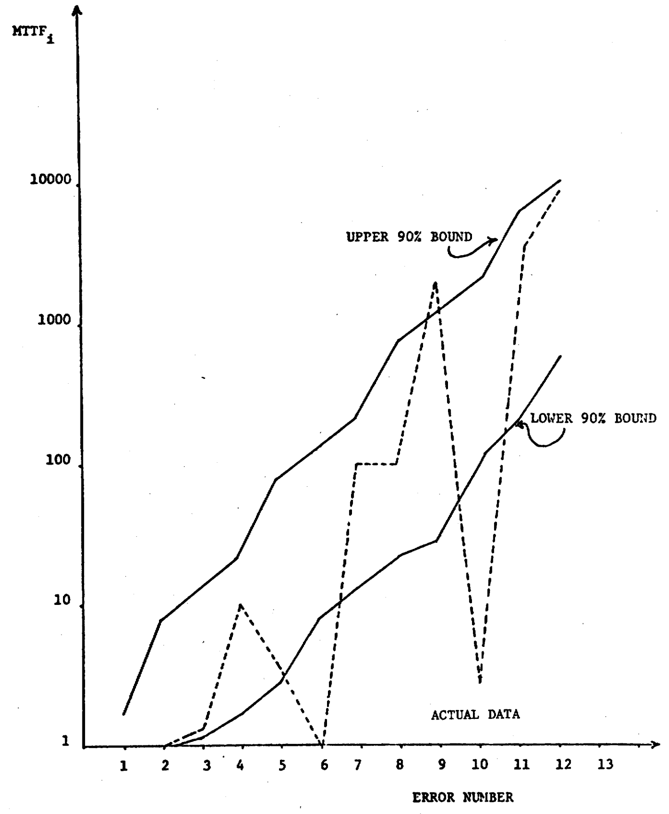


Fig. 9. Upper and lower 90 percent confidence bounds for the time between failures.

$$\frac{j(j+1)}{2} = \sum_{i=1}^{j} \sum_{k=1}^{n_i} \frac{i n_{ik} F_{ik} \hat{a}^i}{1 - F_{ik} \hat{a}^i}$$

where

j = the total number of errors detected;

n_i = the number of transitions from function to function between the detection of the $(i-1)$th and the ith errors;

n_{ik} = the number of times the kth function (in the sequence between the detection of the $(i-1)$th and the ith errors) is tested;

F_{ik} = n, if the kth function (in the sequence between the detection of the $(i-1)$th and the ith errors) is the nth *distinct* function to be tested.

Table VIII-A shows the application of the model to B1, yielding $R(1) = 0.99993$ and $\text{MTTF}_6 = 13903$. The application of the model to the data for B2 shown in Table VIII-B. Here $R(1) = 0.9990$ and $\text{MTTF}_5 = 12896$.

We define the thoroughness (or completeness) of a set of test cases, *irrespective of the test selection strategy*, to be some measure of the confidence in the correctness of the program if it works for the given test cases. That is, it is a measure of our belief in the *a posteriori* correctness of the software [20]. An interesting experimental approach is program mutation due to DeMillo and Lipton [29]. However, this technique is expensive since the number of mutations is combinatorially explosive for programs of realistic sizes. The majority of the mutations have large error sizes and these are

TABLE VIII

A

j	n_j	n_{ji}	f_{ji}	$\frac{1}{\hat{a}}$
1	1	1	1	2.0000
2	1	20	1	4.0000
3	1	20	2	3.6973
4	1	40	2	3.3828
5	2	200 200	1 2	3.4863
6	5	1000 120 300 600 1	2 3 4 5 6	3.9073
R(1) = 0.99993; $MTTF_6$ = 13903				

B

j	n_j	n_{ji}	f_{ji}	$\frac{1}{\hat{a}}$
1	1	1	1	2.0000
2	1	20	1	4.0000
3	2	200 1	1 2	5.2366
4	2	40 20	2 3	4.5700
5	3	1000 120 1	3 4 5	4.8426
6	3	300 600 8	5 6 7	-
R(1) = 0.99990; $MTTF_5$ = 12896				

easily detected. A practical solution is to seed the program with errors, such that the size of the errors is controlled.

We applied the error seeding technique to assess the thoroughness of the set of test cases used in validating programs B1 and B2 in the first phase of the EPRI project. Eight errors were seeded in B1. Errors in expressions were detected by almost all the test cases, implying that arithmetic errors usually have a large size. However, three errors escaped detection. These were of the boundary value and missing control flow type of errors. This indicates that further test cases exercising the ranges of the variables and boundary conditions are necessary.

To summarize, we have developed a software reliability growth model which includes the testing process used during the debugging phase of the project. Error seeding is then used to get an estimate of how well the software has been tested.

VII. Conclusions

The foregoing sections describe an approach for the development and validation of software for critical applications. The results of applying the methodology to the development of a pilot software are also described. The proposed effort has been developed for applications that are small (in terms of the final program size) and critical enough to allow for and justify the large amount of development and validation effort. The dual development of specifications is the most questionable feature (yet the most important in the methodology) for application in a larger scale development, since the development of single set of specifications by itself is already a very significant percentage of the overall effort. We believe, however, that the techniques should be and can be applied to selected portions of the system which are critical to achieve the overall functional objectives.

The project was an attempt at a methodology for automating the development of process control software. The entire effort required nearly $3\frac{1}{2}$ years with the majority of time devoted to the development of the software validation tools. Coding and debugging of the detailed designs required approximately 6 weeks since the identification of the modules was easy from the preliminary design. The algorithms used in the functional requirements were selected jointly by engineers from the Electric Power Research Institute, Science Applications, Inc., and Babcock and Wilcox.

It should be emphasized that the flexible structure of the methodology allowed for ease of revision. Formal specification design, dual specifications, formal validation techniques, the use of powerful automated test tools, and systematic derivation of test cases are some of the highlights of this methodology. Error seeding was used to assess the comprehensibility of the test methods.

A project similar in scope to the one described here has been conducted in Europe jointly by the Technical Research Center of Finland and the OECD Halden Project of Norway [4]. The objective of their project is to develop and test highly reliable programs (computer-based control systems for nuclear reactors). Dual programming is also used and a variety of testing approaches were tried such as simulator generated test data, random test data, seeded errors, etc. A significant departure from the methodology described in Section II is that the specification is not done formally and dual effort starts after the specifications. Furthermore, there is no attempt toward a reliability assessment of the software product. In the Halden Project, two requirements languages were used for the specification and two programming languages in the detailed designs. The results of their project confirm the need for the major elements of our methodology. First, a significant percentage of all errors detected is due to errors or ambiguity in the specifications. A formal specification would be necessary to avoid these problems. Manual translation of specifications to code and validation is also found to be error prone (6.5 percent of total error counts are categorized as "correction errors") and not as effective as automatic test case analysis.

Another issue that we have addressed in detail in the paper is the systematic method of generating test cases in the sense that they will test the functional requirements of the program as compared to others which are chosen after the implementation is done and based on the program structure. Potentially a justifiably complete set of test cases can be derived. Currently, no mechanical methods exist. It is possible, however, to obtain a set of capabilities the system is supposed to deliver and construct test cases to cover all capabilities. But this is an essentially manual procedure.

One of the major problem areas in this project concerns the transformation of the requirements from natural language to a formal requirement specification language such as RSL. The major difficulty involved the validation of the formal specification against the originating natural language requirements. In our project, we attempt to overcome this via dual specification and comparison and manually tracing the formal specification against the originating requirements in English. Powerful methods using natural language analysis techniques may be more appropriate. Also, comparison of two RSL programs at two different levels of abstraction proved to be extremely difficult. Our approach of using high-level symbolic execution was only "semi-automated" and further research into the area of high-level symbolic execution is desirable. In this project, the generation of I/O assertion is a manual error-prone process. This can be observed in the large number of errors caught during the I/O assertion checking phase. We perceive this as a major difficulty in applying the techniques of I/O assertion for development and validation. I/O assertion checking may be difficult to perform on the entire program and for some of the larger and more complex modules. However, for simple modules, assertion checking was easy. In the requirements specification, RSL was used. RSL is still an experimental tool and needs further improvements.

The tools that was used in this project resides in a myriad of machines and requires familiarity with a diverse set of languages. This involves a large amount effort spent by the developers in familiarizing themselves with the development environment. We did not have opportunity to attempt a triprogramming methodology due to cost and complexity factors. In the project there was emphasis in simplicity. This allowed us to concentrate to improve and refine different areas of the methodology one at a time. By adopting this methodology, we believe that we have eliminated most of the errors and reduced the development time of the application programs.

Acknowledgment

Tremendous support and suggestions have come from Dr. A. B. Long of EPRI for the development of the methodology and associated tools discussed in the paper. Several of the ideas were inspired by Dr. C. R. Vick of Systems Control, Inc. and Dr. C. Davis of the Ballistic Missile Defense Advanced Technology Center. We would also like to acknowledge the close collaboration in the project with H. Reeves of the Babcock and Wilcox Company, Dr. S. Saib of General Research Corporation, Dr. E. A. Straker, Dr. R. Downs and Dr. T. Albert of Science Applications, Inc., and our colleagues Dr. S. F. Ho and Dr. H. H. So (now with Bell Laboratories), Dr. C. W. Nam (now with Systems Control, Inc.), J. M. Favaro (now with Siemens, Germany), M. J. Shen (now with Intel Corp.), and K. S. Siyan (now with Rolm Corp.).

References

[1] M. W. Alford *et al.*, "Software requirements engineering methodology," SREP Final Rep., vol. I, TRW Rep. CDRL C005, Huntsville, AL, 1977.

[2] R. E. Barlow and F. Proschan, *Statistical Theory of Reliability and Life Testing.* New York: Holt, Rinehart and Winston, 1975.

[3] F. B. Bastani, "The specification, design, and implementation of an automated test data generator," Master Res. Project Rep., Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, 1978.

[4] G. Dahll and J. Lahti, "Investigation of methods for production and verification of computer programmers with high requirements for reliability," OECD Halden Reactor Project, preliminary version (unpublished paper).

[5] S. A. Davey, "Sample test case," SAI Rep. SAI/SV-UCB-08, Mar. 1979.

[6] W. Ehrenberger and K. Plogert, "Statistical verification of reactor protection software," in *Proc. Int. Symp. Nuclear Power Plant Contr.*, Cannes, paper 39, Apr. 1978.

[7] J. M. Favaro, "A FORTRAN symbolic executor based on MACSYMA," in *Proc. 2nd MACSYMA User's Conf.*, June 1979.

[8] J. R. Foderaro, "Typesetting MACSYMA equations," in *Proc. 2nd MACSYMA User's Conf.*, June 1979.

[9] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," *ACM Comput. Surveys,* vol. 8, Sept. 1976.

[10] S. B. Ho, "A systematic approach to the development and validation of software for critical applications," Ph.D dissertation, Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, Nov. 1978.

[11] A. B. Long *et al.*, "A methodology for the development and validation of critical software for nuclear power plant," in *Proc. 1st Int. Conf. Software Appl.*, Nov. 1977.

[12] W. A. Martin and R. J. Fateman, "The MACSYMA system," in *Proc. 2nd Symp. Symbolic Algebraic Manipulation*, 1972.

[13] Y. R. Mok, "Test plan for a calibrated neutron flux signal device," Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, EPRI RP-961 Project, UCB-ALL-62, 1979.

[14] P. B. Moranda, "Prediction of software reliability during debugging," in *Proc. 1975 Annu. Reliability and Maintainability Symp.*, pp. 327-332.

[15] F. Mosteller and J. W. Tukey, *Data Analysis and Regression: A Second Course in Statistics.* Reading, MA: Addison-Wesley, 1977.

[16] C. W. Nam, "Software preliminary design document for a calibrated neutron flux signal device," Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, EPRI RP-961 Project, UCB-ALL-62, 1979.

[17] K. M. Pitman, "A FORTRAN → LISP translator," in *Proc. 2nd MACSYMA User's Conf.*, June 1979.

[18] C. V. Ramamoorthy and S. F. Ho, "FORTRAN automated code evaluation systems," Electron. Res. Lab. M-466, Univ. California, Berkeley, Aug. 1974.

[19] C. V. Ramamoorthy and F. B. Bastani, "Modeling of the software reliability growth process," in *Proc. COMPSAC 80*, Chicago, IL, Nov. 1980, pp. 161-169.

[20] —, "Software reliability–Status and perspectives," *IEEE Trans. Software Eng.*, to be published.

[21] H. L. Reeves and N. L. Snidow, "Design bases and specification of functional requirements for a calibrated neutron flux signal device," Babcock & Wilcox, Lynchburg, VA, Rep. RP-961, 1978.

[22] H. L. Reeves, "Specification of functional requirements, acceptance test data," Babcock & Wilcox, Rep. SFR-BAW-A3, Apr. 1979.

[23] G. J. Schick and R. W. Wolverton, "An analysis of competing software reliability model," *IEEE Trans. Software Eng.*, vol. SE-4, Mar. 1978.

[24] K. S. Siyan, "The specification, design and implementation of an input/output assertion verifier," Master Res. Project Rep., Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, Dec. 1980.

[25] H. H. So, "An approach to the requirements analysis and specification of large-scale software systems," Ph.D dissertation, Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, Feb. 1979.

[26] L. G. Stucki, "Automated generation of self-metric software," in *Conf. Rec. IEEE Symp. Comput. Software Reliability*, 1973.

[27] TRW Defense and Space Systems Group, "Software reliability study," Rep. 76-2260.1-9-5, TRW, Redondo Beach, CA.

[28] S. H. Sqib, "IFTRAN: Structured programming preprocessors for FORTRAN, IFTRAN-3 user's guide," General Res. Corp., Santa Barbara, CA, Jan. 1978.

[29] R. A. DeMillo, J. Lipton, and F. C. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, Apr. 1978.

[30] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, May 1973.

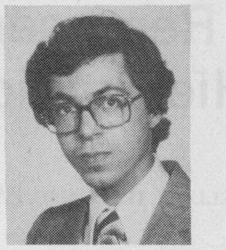
C. V. Ramamoorthy (M'57–SM'76–F'78) received the undergraduate degrees in physics and technology from the University of Madras, India, the M.S. degree and the professional degree of Mechanical Engineer, both from the University of California, Berkeley, and the M.A. and Ph.D. degrees in applied mathematics and computer theory from Harvard University, Cambridge, MA.

He was associated with Honeywell's Electronic Data Processing Division from 1956 to 1971, last as Senior Staff Scientist. He was a Professor in the Department of Electrical Engineering and Computer Sciences at the University of Texas, Austin. Currently, he is a Professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Dr. Ramamoorthy was Chairman of the Education Committee of the IEEE Computer Society and Chairman of the Committee to develop E.C.P.D Accreditation Guidelines for Computer Science and Engineering Degree Programs. He also was the Chairman of the AFIPS Education Committee, a member of the Science and Technology Advisory Group of the U.S. Air Force, and a member of the Technology Advisory Panel of Ballistic Missile Defense (U.S. Army). He is currently the Vice President for Education of the IEEE Computer Society.

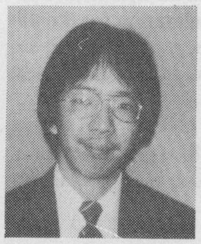
Yu-King R. Mok (S'75–M'80) received the B.S. degree in electrical engineering from the University of California, Davis, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley.

He is currently with Bell Laboratories, Naperville, IL. His current interests are the area of software development and validation, distributed systems, computer networks, and communication.



Farokh B. Bastani received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, India, in 1977, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1978 and 1980, respectively.

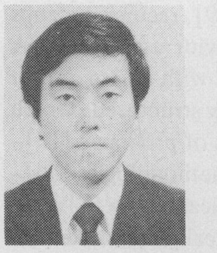
He joined the University of Houston, Houston, TX, in 1980 where he is currently Assistant Professor of Computer Science. His research interests include developing a design methodology and quality assessment techniques for large-scale computer systems.



Gene H. Chin (M'79) received the B.A. degree in computer science from the University of California, Berkeley, in 1981.

He is currently a Software Engineer with the Information Networks Division, Hewlett-Packard Corporation, Cupertino, CA, where he is working on software quality assurance and testing, reliability engineering, and evaluation. His research interests include software design methodology, testing, query languages, and database management.

Mr. Chin is a member of the Association for Computing Machinery.



Keiichi Suzuki (M'78) received the B.A. degree in pure and applied sciences and the M.S. degree in coordinated science from the University of Tokyo, Tokyo, Japan, and the M.S. degree in computer science from the University of California, Berkeley.

He was with Hitachi Software Engineering Company, Yokohama, Japan. He is currently a Software Engineer with Intel Corporation, Santa Clara, CA, where he is working on software evaluation.

Mr. Suzuki is a member of the Association for Computing Machinery.