AD-A047 653 CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB F/G 9/2
AUTOMATED EVALUATION SYSTEMS.(U)
SEP 77 C V RAMAMOORTHY N00014-75-C-0485
UNCLASSIFIED NL

| OF |
AD
A047653

END
DATE
FILMED
1 -78
DDC

(12) D.S.

AUTOMATED EVALUATION SYSTEMS

Final Report

C. V. Ramamoorthy

November 15, 1974 - September 14, 1977

Office of Naval Research

Contract: N00014-75-C-0485

Electronics Research Laboratory
College of Engineering
University of California, Berkeley
Berkeley, California 94720

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Automated Evaluation Systems,		5. TYPE OF REPORT & PERIOD COVERED (9) Final rept. 15 Nov 74 - 14 Sep 77
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) C. V. Ramamoorthy		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0485
9. PERFORMING ORGANIZATION NAME AND ADDRESS Electronics Research Laboratory, University of California Berkeley, California 94720		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (11) 14 Sep 77
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, Virginia 22217		12. REPORT DATE (12) 24 P.
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

final report: 15 November 1974 - 14 September 1977

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

software development
automated testing tools
reliability
large-scale software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The overall objective of the research has been to investigate software development process, automated testing tools, and development techniques for improving reliability of large-scale software systems. In order to accomplish this objective, a preliminary survey on reliability and integrity of large computer programs has been conducted and a scheme has been proposed as a reasonable approach to developing reliable large-scale software.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

127550

1. Introduction

This report summarizes the findings of the research supported by the U. S. Office of Naval Research Office under the contract N00014-75-C-0485 for three years, November 15, 1974 through September 14, 1977. Most of the results reported here have been published in technical papers, Ph.d. dissertations, and M.S. research reports cited in the References. It is gratefully noted that ten graduate students have been supported by the grant to pursue advanced degrees at the University of California, Berkeley.

The overall objective of the research has been to investigate software development process, automated testing tools, and development techniques for improving reliability of large-scale software systems. In order to accomplish this objective, a preliminary survey on reliability and integrity of large computer programs has been conducted and the following scheme has been proposed as a reasonable approach to developing reliable large-scale software [1]:

(1) Specification of the system.
(2) Design of the structure, decomposition, and modularlization of the system, with the specification of each module.
(3) Coding of the system in a suitable programming language.
(4) Debugging, integration, and check out of the system.
(5) Software evaluation and partial validation with the help of automated tools.

(6) Software fail-safe, fail-secure instrumentation.

(7) Validation of protection and security measures.

In order to accomplish the overall objective along this scheme, the following research areas have been set and substantial results have been obtained in each area:

(1) Design and development of the FORTRAN Automated Code Evaluation System (FACES).

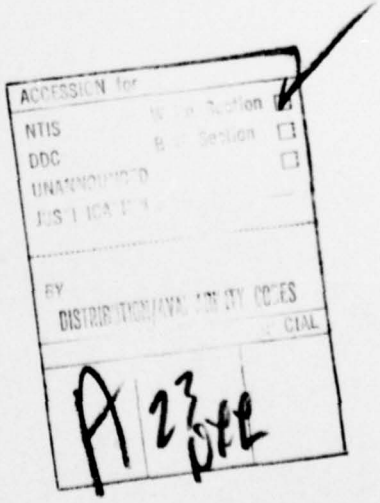
(2) Investigations in software requirements and specifications.

(3) Software performance evaluation based on dynamic analysis.

(4) Design and development of an adaptive compiler.

These sub-objectives are intended to improve reliability of software systems in production, testing, and maintenance stages.

We shall briefly discuss our findings in these research areas in the following four sections, one section for each area in the above mentioned order. A summary section will be found at the end of this report.



2. Design and development of the FORTRAN Automated Code Evaluation System (FACES)

Significant progress has been made in the proof of program correctness techniques in recent years. However, this approach is not yet feasible for large scale systems because of complexity and high cost of implementation. Hence, program analysis, testing, and evaluation are still among the most effective means for attaining and assuring software quality in practice. The feasibility of improving software reliability and quality through the use of automated software tool has been investigated. [2] Automated software evaluation can be defined as the process of checking presence or absence of certain software attributes (characteristics) throughout the life-cycle of a software system for ascertaining software quality. The development of the FACES system has undergone the following phases. First of all, the software development process was analyzed to identify areas where tools are applicable and the fundamental notions underlying these software tools were identified. Second, basic techniques for program analysis were introduced, i.e. static and dynamic analysis techniques. Static analysis examines the static source code to uncover errors and inconsistencies among modules. Dynamic analysis examines run time program behavior. Third, three major categories of software tools (i.e. program source code analyzers, runtime analysis tools, and program testing aids) and their limitations were analyzed in detail. Finally, based on these analyses the design and implementation of the FORTRAN Automated Code Evaluation System (FACES) was conducted.

FACES [2,3,4] is a tool designed for assisting the development, testing, modification, and maintenance of FORTRAN programs. FACES is capable of performing (1) source code analysis, (2) run-time analysis, and (3) automated test case generation of the input FORTRAN programs. These functions are performed by the corresponding Automated Interrogation Routine (AIR), the Dynamic Analyzer, and the Test Case Generator (CASEGEN) as depicted in Figure 1. A major design objective of FACES is flexibility. FACES is provided with a language preprocessor which transforms the source information to be analyzed to an appropriate tabular representation in a data base. The tabular representation is a transactional format of program operations which is relatively free of implicit language properties. Using preprocessors, FACES can be adapted to validate programs written in different programming languages, e.g. COBOL programs.

In addition, the construction of the data base is intended to provide a convenient means of retrieving various program attributes. The philosophy is to relieve the analyzer of the tedious process of examining the source information required to validate the program. Furthermore, the data base forms the basis for other validation techniques, thus avoiding duplication of efforts. The data base generated by the language preprocessor consists of three main tables: (1) Symbolic table in which symbolic elements of the input information such as variable names, labels, subprogram names, etc. are catalogued. (2) Use table which records how and where in the input source information the symbolic element is referenced, e.g. where calls appear to subroutines and what are the variables involved in an assignment statement.

The use table allows questions concerning the usage of variables to be answered easily. (3) Node table which records the program flow structure. The structure of a program is modelled as a directed graph with each statement represented by a node. The node table is essential for structural analyses. These tables are not independent of each other and they are usually linked together by logical assertions so that the movement from one table to another causes a logically related item to be addressed.

Automated Interrogation Routine (AIR) interprets queries and automatically searches the data base for the specified language constructs. The types of queries include (A) software quality checks, (B) specific user requests, and (C) documentation production. Software quality inspection queries are a set of application independent queries used to improve the reliability and quality of software systems. These queries perform (1) error prone construct identifications, (2) interface checks, (3) program structure tests, (4) coding standard checkes, and (5) uninitialized variable check. Specific user requests queries allow users to specify queries and then automatically searches the data base for the user-specified language constructs. Users may specify the area of search to be the entire software system or an individual routine. Document generation extracts information from the data base to generate various cross-reference tables and program structure diagrams. Some of these are variable versus statement cross reference table, subroutine calling sequence table, COMMON block versus subroutine cross reference table, and program graph.

The Dynamic Analyzer performs dynamic analysis which is

basically a process of software testing consisting of driving the program with the test input, observing the run time program behavior and evaluating the outputs. It carries out both validation functions such as error diagnosis and performance evaluation. For this purpose, supplemental codes (monitors) are inserted inside the target source program. Monitors perform frequency monitoring, variable value tracing, and execution path tracing. Capabilities for checking user specified assertions have also been implemented. Assertion checks can be viewed as intentional software redundancy to improve program reliability. Assertions can be any legal FORTRAN conditional expressions inserted into the program source code comments. Along with the development of the Dynamic Analyzer, theoretical work has been directed into finding a minimal set of monitors and their locations [5,6,7,8] such that the amount of overhead (extra storage and time) incurred during program execution is minimized.

The Automated Test Case Generator (CASEGEN) [9,10,11] has been designed and built to generate test data automatically for testing FORTRAN programs and consists of three major components: (1) Path Generator, (2) Path Constraint Generator, and (3) Test Data Generator. A schematic diagram of CASEGEN is given in Figure 2. The Path Generator accesses the common data base and generates a near minimal set of paths to cover all edges. The theoretical basis for finding a minimal set of paths has been investigated and the results influenced the design and implementation of CASEGEN. [11] To allow easy analysis of program structure, a program is modelled as a directed graph in which nodes represent program segments and branches represent program flow of

control. With this model an algorithm of linear complexity for finding a subset of paths needed to cover all branches has been developed. [12] Given the paths, the Path Constraint Generator uses forward substitution to synthesize the appropriate constraints. Loops and array references are treated with special concern. [13] The output of the Path Constraint Generator is a set of equality and inequality constraints on the input variables. The Test Data Generator systematically assigns values to the input variables until all the constraints are satisfied, using random numbers to generate inputs. [11]

During the development of FACES, it became clear that an interactive system would reduce the complexity and increase the efficiency of the testing process by enabling programmers to guide FACES in its testing decisions. With this in mind, the design of a prototype of the Interactive Software Evaluation System (ISES) has been completed. [11] ISES augments FACES to provide an interactive environment in which a large software system can be developed, validated, tested, and maintained. ISES allows users to edit and then submit programs to FACES for analysis in an interactive mode.

3. Software Requirements and Specifications

The problem of software reliability can be attacked in two different, but complementing ways, i.e. by developing better methods of software construction and by development of better validation techniques and tools. A software specification system has its greatest contribution to more reliable software right at the beginning. The basic objective is to allow system designers to say what is intended to be done precisely so that programs can be coded in a more efficient way. Moreover, frequent checkings are possible throughput the development period to assure a certain degree of achievement at any point of time during development.

A software specification system should be formal enough for a variety of mechanical checkings such that there is a definite degree of achievement as the design proceeds. The most important ingredient of a specification system is the specification language. A preliminary design of such a language has been conducted. [14] Its main application area is in large-scale real-time software. The following specifications are included in the language: (a) data specifications, (b) process specifications, (c) interaction specifications, and (d) performance requirement specifications. These four categories together cover most of the areas in which software errors (especially during the design phase) are discovered. The problem of the language processor have also been briefly considered to the extent that the amount of required processing becomes fairly apparent.

The principles underlying and a system for computer entry,

representation, and analysis of a specification language describing a target system has been developed. [10] The specification language is a top-down hierarchical module-oriented design, with a particular emphasis on abstraction as a mechanism for organizing the functions of the system and for delaying the description of low-level details. Specifications for a module are grouped into three basic categories: interfaces, covering those aspects of a module seen from the outside; attributes, defining the operational constraints of a module functioning within the environment; and functional specifications, providing constructs describing the functional operation of a module ranging from essentially non-procedural forms to a full capability for defining sequences, iteration, and conditional expressions. The language is interactively oriented, designed for use with sophisticated software aids that provide for an iterative, step-by-step progression toward the completion of a target system. A relational database is used for the internal representation of specification constructs. Analysis techniques include the use of a formal graph model, modular analysis, and cross checks of internally represented information. The development of a specification/analysis system has progressed to the state of a prototype implementation design. [15] The nature of the system as seen by the user has been clearly identified. Extensive information and details on the syntax, semantics, representation, and analysis of the specification constructs have been provided to enable the future implementation of a working system.

4. Software Performance Evaluation Based on Dynamic Analysis [6]

An approach for evaluating and measuring software reliability has been developed by exploiting the structural properties of programs. In this approach, the program structure is represented by a directed graph consisting of a set of nodes and a set of directed arcs. A node can be defined as a computational task which consists of a linear sequence of program instructions having one entry point and one exit point. Each directed arc represents a possible transfer of control from the exit point of a node to the entry point of another. A node may have several arcs directed to its entry point or from its exit. This graph model is conceptually simple, origination from flow charts. This model has the advantage of the ability to represent the structure of a program at any level of abstraction in a clear simple way easily interpreted by human programmers. A unified theory of program frequency measurement, which is composed of arc frequency measurement and path frequency counting, has been proposed to collect information about the dynamic behavior of programs. The optimal sites for such instrumentation has been given, which enable us to minimize the measurement overhead [5,6]. Other measurement of the behavior of critical variables in the program has also presented. All these instrumentation tools can be inserted automatically after an automated analysis of the program. Automation is essential for these tools to be applicable. Otherwise, programmers will be reluctant to use them due to inconvenience and unreliability.

The philosophy of structural program testing emphasizes the

role of program structure in the design of effective tests. Given a set of valid inputs, the program generates a dynamic process to carry out the appropriate activities that the programmer has prescribed. The program is correct if and only if the dynamic processes it generates are correct. When a process terminates, it will correspond to a path (not necessarily a simple path) for the entry node to the exit node of the program graph. The number of the legal processes that can be generated by a program is an astronomical number. An effective approach to simplify this problem is to group processes into equivalence classes by neglecting detail differences among the processes in the same class. In termes of the graph model the formation of equivalence classes corresponds to the contraction of groups of nodes into a larger node. This formation is carried out automatically or with the help of human programmers. Structural program testing requires the examination of the program graph and the selection of a particular path (hence the corresponding process) to be tested every time. Depending on different testing requirements criteria, the path selected to be tested may be different. Designing tests in this fashion has several advantages such as allowing us to distribute our testing effort to test different parts of the program in different tests and ease of generating the reasonableness checks on the test output and of verifying that the test data really test the process we have in mind. In order to activate a particular process test input data should be designed to "sensitize" the corresponding path in the graph. A simple method has been presented for this sensitization. A strategy called "rollback testing" similar to rollback and recovery in fault-

tolerant computing systems has been proposed for designing effective tests. Instrumentation for structural analysis also gives us valuable information to rewrite parts of a program to improve execution efficiency of the program.

Some quantitative measures for the important parameters of the quality of programs, i.e. reliability, maintenability, and availability, have been derived. The reliability of a program is expressed in terms of the quality of the service the program provides to the user. It was shown that software reliability can be modelled by a simple Markov model. Although this model is too simple for accurate prediction of the quality of programs, the model indicates how to improve reliability of programs effectively. A program can be said more maintainable if the "ripple effect" caused by the interaction among modules is small. Maintenability of a program has been modelled by a graph model. As in the case of the reliability model, this simple model is not intended to be an accurate prediction of the expected number of changes for every maintenance.

5. Adaptable Compiler [16,17,18]

Software tools for testing and debugging large programs such as FACES discussed in the previous section are among the facilities available to users for the development of large programs. The cost incurred in maintaining existing software is also immense in an always changing environment and usually surpasses the original development cost. One approach to reduce maintenance cost of programs is to enhance software portability and adaptability, which are most influenced by the application, the host language, and the system support (i.e. compilers, operating systems, hardware organization).

With the decreasing hardware costs, application of minicomputers is becoming common. A limiting factor to small machine application is the development of appropriate software systems having good portability and adaptability, since their properties vary diversely and individual capabilities are limited. Therefore, a software system development tool for minicomputer type architecture, called the Multi-Mini Computer Compiler (MMCC) has been investigated as a solution to this problem. Specific areas studied include: (1) design of a machine independent, but minicomputer-oriented intermediate language, (2) formalizing the description of the minicomputer programmable resources, (3) automatic translation of the intermediate language instructions into individual minicomputer assembly instructions.

In MMCC, the high-level application language is first translated into the minicomputer-oriented intermediate language, called the Meta Assembly Language (MAL). The MAL is then

translated into the symbolic assembly language of the minicomputer whose description is provided as parametric input to MMCC. A typical target machine description includes processor organization, word size, addressing schemes and the instruction set. A finite state model of code generation has been built into the automatic code generating system. This model is automatically reconfigured for each new target machine according to its description. MMCC generates code by mapping the MAL instructions, line by line to those of the target machine. Code generation process is directed by the finite state transitions and the description of target machine programmable resources.

This compiler is highly adaptable to the execution environment since the object code compatible with the target machine will be produced from a description of the intended target machine's resources. Development of MMCC brings along several advantages not presently available in conventional compilers. MMCC increases the pool of available qualified personnel, promotes uniformity of software systems, reduces education time when personnel shifts are required, and permits more graceful transition among machines. The designer will be able to separate development activities associated with logical system functions from target machine operational quirks. This added flexibility allows the software and hardware designer to proceed concurrently. Also, automated tools may be developed to test each activity in the absence of the other, thus resulting in speed-up in production time.

The Meta Assembler for TI960A and for HP 2100A minicomputers have been implemented on the CDC 6400, using PASCAL as the host

language. The experience with the meta assembler shows, provided the user of the system is familiar with the target machine, that the MAL abstract machine and the formalism which describes the machine features (i.e. the machine specification language), the efforts required to adapt a new target environment are surprisingly small.

6. Summary

This report summarizes the research findings obtained under the contract N00014-75-C-0485. The overall objective of improving reliability of large-scale software systems has been divided into four sub-objectives, i.e. development of the FORTRAN Automated Code Evaluation System, software requirements and specifications, software performance evaluation based on dynamic analysis, and adaptive compiler. The FORTRAN Automated Code Evaluation System (FACES) has been implemented based on the results of several theoretical studies such as optimization of monitors and minimal set of program execution paths. In the study of software requirements and specifications, a specification/analysis system has been developed to the prototype implementation design. A structural theory has been proposed and applied for software performance evaluation based on dynamic analysis. Finally, a software system development tool for minicomputer type architecture has been investigated as a solution to improve adaptability of software systems.

Personnels Supported (all graduate students in Berkeley)

I. Ph.D.

(1) W. T. Chen 1976

(2) R. C. Cheung 1974

(3) A. P. Conn 1977

(4) I. Friedmann 1977

(5) V. Gligor 1976

(6) P. Jahanian 1977

(7) W. Y. Han 1976

(8) K. H. Kim 1974

II. M.S.

(9) S. F. Ho 1976

(10) H. H. So 1975

References

[1] C. V. Ramamoorthy, R. C. Cheung, and K. H. Kim, "Reliability and Integrity of Large Computer Programs," Info. Tech. Rep. Computing System Reliability, 1974; also Electronics Research Laboratory, University of California, Berkeley, ERL-M430.

[2] C. V. Ramamoorthy and S. F. Ho, "Testing Software with Automated Software Evaluation Systems," Proc. International Conference on Reliable Software, 1975. Also, IEEE Transactions on Software Engineering, March, 1975.

[3] S. F. Ho, "Automated Software Testing and Evaluation," Master Research Report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1976.

[4] C. V. Ramamoorthy and S. F. Ho, "FORTRAN Automated Code Evaluation System," Memorandom No. ERL M-466, Electronics Research Laboratory, University of California, Berkeley, August 1974.

[5] C. V. Ramamoorthy and R. C. Cheung, "Optimal Measurement of Program Path Frequencies and Its Applications," International Federation of Automatic Control, Boston, August, 1975.

[6] R. C. Cheung, "A Structural Theory for Improving Software Reliability," Ph.D. Dissertation, Department of Electrical Engineering and computer Sciences, University of California, Berkeley, 1974.

[7] C. V. Ramamoorthy and K. H. Kim, "Software Monitors Aiding Systematic Testing and Their Optimal Placement," The First National Conference of Software Engineering, Washington, September 1975.

[8] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, "Optimal Placement of Software Monitors Aiding Systematic Testing," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, December 1975.

[9] W. T. Chen and C. V. Ramamoorthy, "Toward Automation of Test Data Generation," International Computer Symposium, 1975, Taipei, taiwan, Republic of China, August, 1975.

[10] C. V. Ramamoorthy, W. T. Chen, W. Y. Han, and S. F. Ho, "Techniques for Automated Test Data Generation," Proceedings of the Ninth Annual ASILOMAR Conference on Circuit Systems and Computers, November 1975.

[11] W. T. Chen, "Toward the Design of an Interactive Software Evaluation System," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of

California, Berkeley, February 1976.

[12] Y. W. Han, "Computer Reliability - Hardware and Software," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, February 1976.

[13] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," Proceedings of the Second International conference on Software Engineering, 1976.

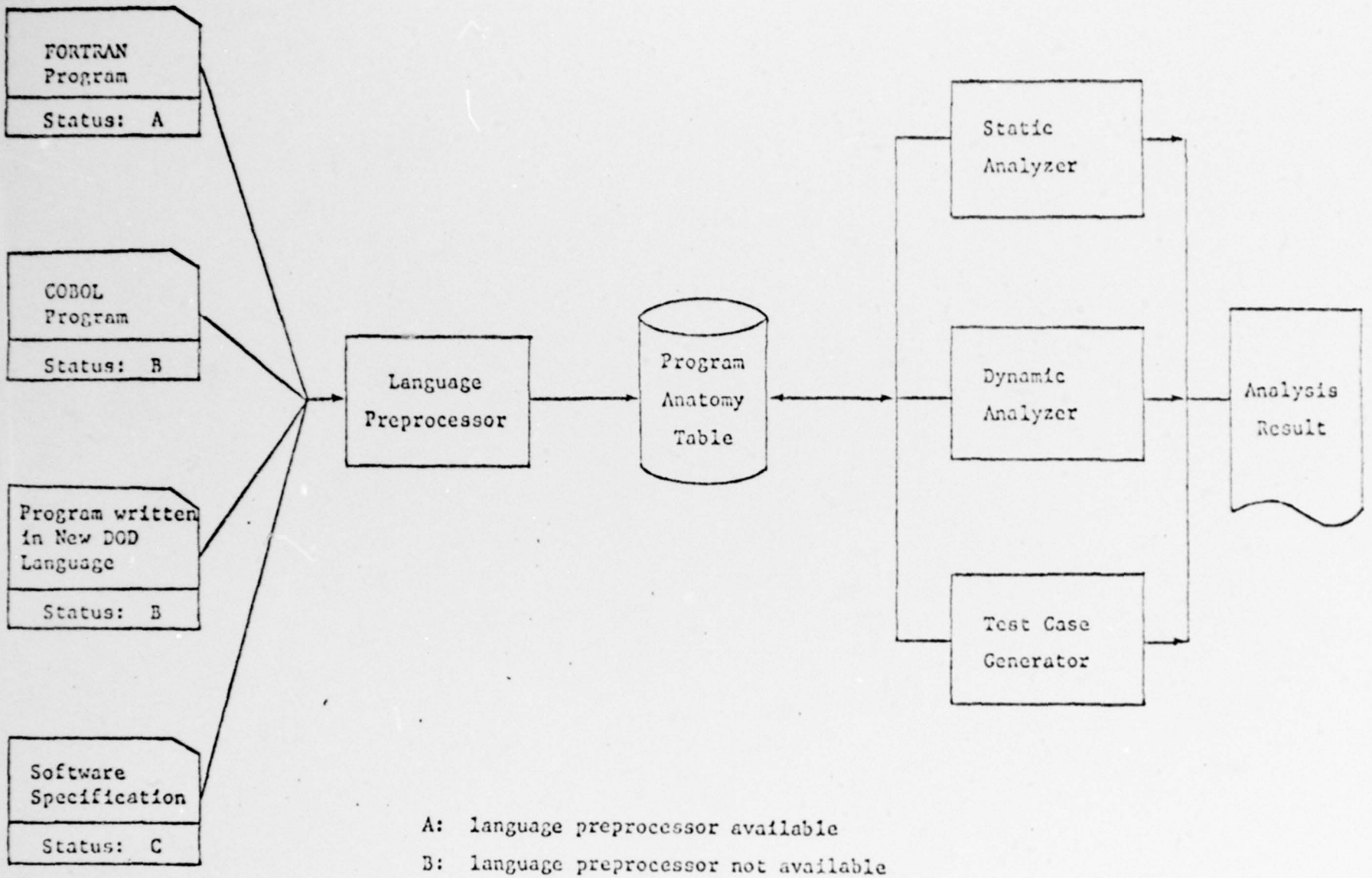
[14] H. H. So, "Preliminary Considerations of Software Specifications," Master Research Report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1975.

[15] A. P. Conn, "Specification of Reliable Large-Scale Software Systems," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1977.

[16] P. Jahanian, "Design and Implementation of an Adaptable Complier for Minicomputer Type Architecture," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1977.

[17] C. V. Ramamoorthy and P. Jahanian, "Formalizing the Specification of Target Machines for Compiler Adaptability Enhancement," Proceedings of the Symposium on Computer Software Engineering, Polytechnic Institute of New York, April 20-22, 1976.

[18] C. V. Ramamoorthy and P. Jahanian, "Formalizing Code Generation in the Multi-MiniComputer Compiler," Proceedings of the Sagamore Computer Conference on Parallel Processing, August 1975.



A: language preprocessor available

B: language preprocessor not available

C: language preprocessor partially available

Figure 1. FACES

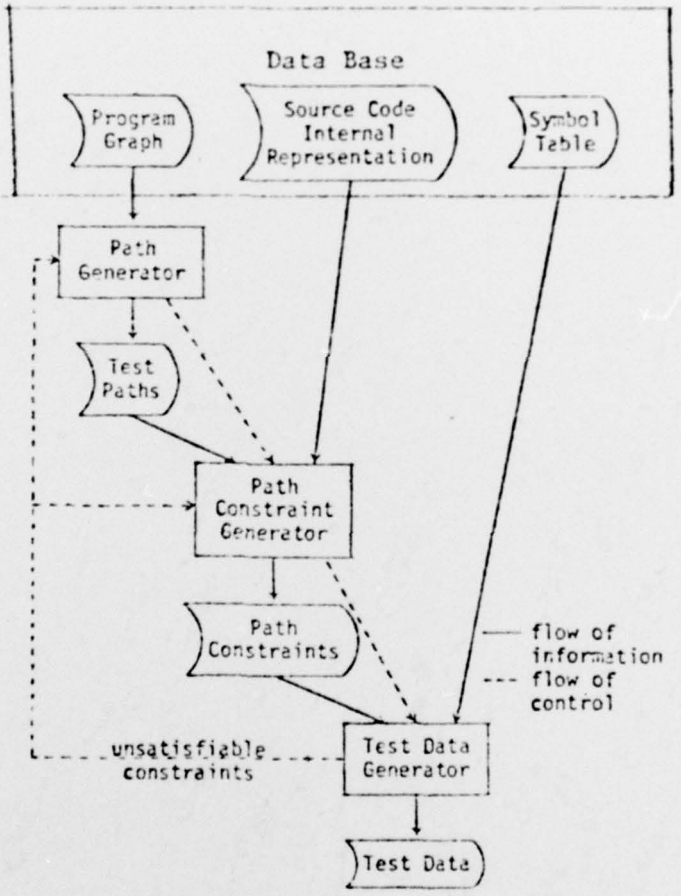


Figure 2. CASEGEN