B

A048057

AD A107719

UNITED STATES ARMY

COMPUTER SYSTEMS COMMAND

A

THE VERIFICATION OF COBOL PROGRAMS

FORT BELVOIR, VIRGINIA 22060

81 [illegible]

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) UNCLASSIFIED

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO AD-A107 717	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE VERIFICATION OF COBOL PROGRAMS		5. TYPE OF REPORT & PERIOD COVERED Final 15 January 1975 to ~~15 October 1976~~
		6. PERFORMING ORG. REPORT NUMBER SRI 3967-2
7. AUTHOR(s) L. Robinson, M. W. Green, R. E. Shostak and J. M. Spitzen		8. CONTRACT OR GRANT NUMBER(s) DAHC04-75-C-0011
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford Research Institute 333 Ravenswood Avenue Menlo Park, California 94025		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Computer Systems Command Advanced Technology Directorate Fort Belvoir, Virginia 22060		12. REPORT DATE 31 March 1976 / 13. NO. OF PAGES 196
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) Dr. Jimmie Suttle U. S. Army Research Office Box CM, Duke Station ~~Durham, North Carolina 27706~~		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Distribution Unlimited

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

18. SUPPLEMENTARY NOTES

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

Program verification, COBOL, structured programming

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

This report describes an initial study concerning the feasibility of proving the correctness (i.e., verification) of COBOL programs. The report contains: (1) a study of the COBOL language as related to verification, (2) the syntax and semantics of a subset of COBOL 74 in which to perform experimental verification, (3) design of a system to accomplish COBOL verification in the desired subset, and (4) proof of a sample COBOL program. The conclusion of the report is that COBOL verification is indeed feasible, but must be further engineered if it is to be cost-effective.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

3325

March 1976

Final Report
Covering the Period 15 January 1975 to 31 January 1976

THE VERIFICATION OF COBOL PROGRAMS

By: L. Robinson (Project Leader)
M. W. Green
R. E. Shostak
J. M. Spitzen

Prepared for:

U.S. ARMY COMPUTER SYSTEMS COMMAND
FORT BELVOIR, VIRGINIA 22060

Contract DAHC-04-75-0011

STANFORD RESEARCH INSTITUTE
333 RAVENSWOOD AVENUE
MENLO PARK, CALIFORNIA

A

FOREWORD

This report was prepared in support of the US Army Computer Systems Command Research and Development Program. The report was prepared by Stanford Research Institute under Contract Number DAHC-04-75-0011.

This Technical Report has been reviewed and is approved.

John Staudhammer
DR. JOHN STAUDHAMMER
Technical Advisor
US Army Computer Systems Command

[illegible]
GIORA PELLED
Director, Advanced Technology
US Army Computer Systems Command

Jonathan B. Price
JONATHAN B. PRICE, 1LT SC
Task Action Officer
US Army Computer Systems Command

ABSTRACT

This report describes an initial study concerning the feasibility of proving the correctness (i.e., verification) of COBOL programs. The report contains:

(1) A study of the COBOL language as related to verification.

(2) The syntax and semantics of a subset of COBOL 74 in which to perform experimental verification.

(3) Design of a system to accomplish COBOL verification in the desired subset.

(4) Proof of a sample COBOL program.

The conclusion of the report is that COBOL verification is indeed feasible, but must be further engineered if it is to be cost-effective.

PREFACE

For readers who want only a summary of COBOL verification work, we suggest reading Sections I, II, III, and VII. The introductions and conclusions to the remaining sections may also be helpful. A glossary is provided that defines terms pertinent to program verification. We do not suggest reading the Appendix unless the reader is interested in the inner workings of the COBOL verification system and has some background in LISP.

The authors acknowledge the assistance of Jack Goldberg and Karl N. Levitt in the management of this project, and the technical assistance provided by Robert S. Boyer and Bernard Elspas.

CONTENTS

PRECEDING PAGE BLANK-NOT FI

ILLUSTRATIONS

TABLES

I INTRODUCTION

A. Introduction

The goal of this project has been to study the feasibility of formal verification of COBOL programs. To do that, we have chosen a subset of the COBOL and built part of a system to verify programs written in that subset. We have shown that it is possible to verify COBOL programs, but that there are many problems yet to be resolved to make COBOL verification cost-effective. In this report, we outline some of these problems and, in some cases, propose solutions.

Program verification is not yet applicable to the production of software. Although most of the theoretical problems have been solved, tools must be developed to reduce the volume of material that programmers now must process in program verification. Among the tools needed are well-engineered interactive aids and new programming languages.

In our work we have:

(1) Analyzed COBOL with respect to verification.

(2) Selected a subset of COBOL for verification.

(3) Designed an assertion language for formally describing the intent of a COBOL program.

(4) Designed a system using the INTERLISP language for generating verification conditions for COBOL programs.

(5) Discussed the implications of structured programming on COBOL verification.

(6) Presented an example of a verified COBOL program of modest size ($\leq$100 lines in the PROCEDURE DIVISION).

In this section, we discuss the goals and background of the project, the theory of program verification, the programs verification for "real" languages, and the structure of COBOL.

In Section II, we present a detailed analysis of COBOL in relation to program verification, including general comments about including or excluding particular language features in the COBOL subset for verification (hereinafter called the CSV).

In Section III, we describe the structure of the verification system designed for this project.

In Section IV, we formally describe the syntax of the CSV.

In Section V, we describe the assertion language and some rules of inference for deduction.

In Section VI, we describe the semantics of the CSV.

In Section VIII, we present an example of a fully verified COBOL program.

In Section IX, we state the conclusions of the project.

The documented code for the programs that we have designed to assist in COBOL verification is in the Appendix.

B. Background and Project Goals

The goal of program verification--"verification" throughout this report refers to formal mathematical proof-- is to make programs more reliable. In our work, we make several assumptions. First, a programming language is a formal medium for expressing solutions to certain types of problems. Because of this formality, a program can be analyzed logically. Second, a program (especially a large one) is not necessarily a good medium for stating the problem to be solved. A program states how a problem is to be solved, not what the problem is.

Program verification is not yet a viable tool to improve software reliability because:

(1) The programs are too complex.

(2) The assertions are too complex.

(3) The programs have been written in a programming language that is not amenable to formal description.

The means of solving the first two problems is that of structuring.[1,2] The primary goal of the work described in this report is to solve the third problem. The other two issues are discussed in this report, but not emphasized.

Even though program verification may be difficult, we believe that its usefulness can be increased to approach acceptable limits. The answer lies in the seriousness of the software problem and in its inherent nature. Almost every nontrivial program has some logical flaw; many commercial programs have so many "bugs" that they do not provide the service promised by the vendor. Some programs contain undetected but potentially damaging errors even after being tested and debugged for some time.

The inherent nature of the problem is that informal standards (English-language specifications) cannot be used to guarantee the correctness of a program. As the technology matures, some formal notation is usually developed so that standards can be stated unambiguously and the product can be checked systematically against its standards. A doctor's prescription is a simple example of such a formal medium that is typically not checked. Whether or not this checking is done in every case is usually a matter of cost-effectiveness.

The same thing is true for program verification. We must develop an unambiguous language for stating standards (i.e., an assertion language) and techniques for systematic checking (i.e., formal verification). We can then imagine a scenario in which assertions are written out, but might or might not be formally checked against the program, depending on cost-effectiveness. But at least some checking mechanism exists, so that if the application is critical, the cost can be incurred. Thus, it is useful to do research in verification techniques and to write assertions for programs, even if verification is not attempted. An added benefit of this research is to distinguish language features that simplify analysis of programs, even if that analysis stops short of proof.

This work is an attempt (the first that we know of) to apply formal verification techniques to COBOL. It is also one of the few attempts to use verification techniques for any of the commonly used languages.

C. Program Verification--Theory

The idea of program verification goes back almost as far as programming itself: it was first discussed by von Neumann and Goldstine.[3] The basic idea is that there is a state that models some external phenomenon (e.g., differential equations, matrices, payroll records). The state can be represented by the contents of core memory, the contents of files, or program variables (at a more abstract level). There is also a set of elementary operations that change the state. Examples of elementary operations are machine instructions or statements in higher-level programming languages. A program defines a (possibly infinite) set of sequences of elementary operations. When a program is executed, only one sequence of elementary operations is performed. The selection of one sequence out of the set of sequences defined by the program is determined by the state just before the program is executed called the "initial state." Thus, a program is a function from states to sequences of operations. If the program terminates, the state just after termination is called the "final state."

The user of a program is interested in knowing what the final state will be for a given initial state of the program. Ideally, he will have a specification, which expresses a mapping from initial states to final states. It is not immediately obvious whether a program (also a mapping from states to states) is consistent with the specification. Consistency between a specification and a program is often called "program correctness." Program verification is a set of techniques for proving this consistency. Floyd[4] and Naur[5] first described this method of verification. The specification consists of a statement of the properties that the initial state must have (the input assertion), and a statement of the relation between the initial state and the final state (the output assertion). Both input and output assertions are stated as logical predicates.

The effects of each of the elementary operations on the state must also be formally described (input and output assertions for these operations are useful for this purpose). The control operations, which do not in themselves affect the state, must also be axiomatized. A program may in a small number of statements describe a large (possibly infinite) sequence of operations. To achieve verification, inductive assertions, which break the program's flowchart up into finite sequences of operations, must be associated with each of the loops of the program.

Floyd's method is used for proving partial correctness of programs. A partially correct program is consistent with its assertions only if it terminates. Termination of a program can be proved separately. Given input and output assertions, program text (with inductive assertions), and the definition of the elementary operations, a set of formulas in first-order logic can be constructed whose validity is equivalent to the partial correctness of the program. These formulas are called "verification conditions." A software system that accepts as input the program to be verified (with input, output, and inductive assertions) is called a "verification condition generator."[6,7] Verification conditions can be proved by hand, or can be input to a deductive system or automatic theorem prover, which attempts to generate a proof. In general, deductive systems are inadequate for proving verification conditions by completely automatic means, and many systems are equipped with interactive facilities to allow users to guide the proof. Deductive systems with interactive facilities are also called "semiautomatic verification systems." A diagram of a program verification system is shown in Figure I-1.

The application of formal techniques to a particular programming language environment is often a matter of style. The verification condition generator incorporates most of the language-dependent features, because it must translate assertions and statements in the programming language into expressions in predicate calculus. Some verification condition generators are based on a particular semantic description of a language. For example, a verification condition generator for PASCAL (London, Luckham, and Igarashi[7]) is based on the axiomatic description of PASCAL by Hoare and Wirth.[8]

Several issues have not been addressed by the mainstream of program verification: The first issue, termination, has been addressed by several researchers.[6,9,10] It can be treated either with or separately from the issue of partial correctness. It is important to formalize two other issues -- run-time errors and validity of input data -- if verification is to lead to software reliability. All three of these issues have been grouped, to some extent, into a property called "clean termination."[11] Although these issues are important, they are not considered during this work, which limits itself to the basic issues of partial correctness for COBOL programs.

D. Program Verification for "Real Languages"

In this subsection, we attempt to define the concept of a real programming language by enumerating some of its properties. Particularly important are the properties of semantic cleanliness and syntactic size of real languages. We also describe some special properties of programs for which proof is particularly important when dealing with real languages.

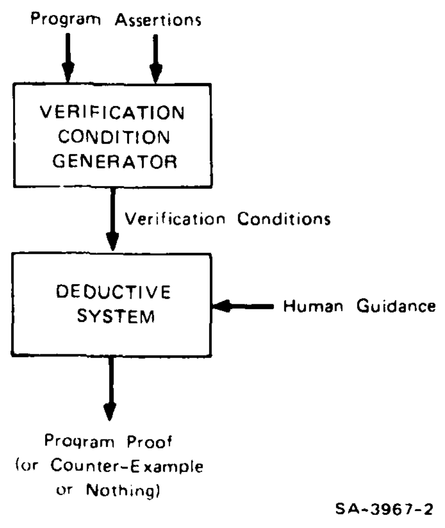


FIGURE I-1 DIAGRAM OF A SEMIAUTOMATIC PROGRAM VERIFICATION SYSTEM

COBOL is a member of the set of "real" programming languages, which are those that are widely used in many applications and for which standards exist. Real languages are usually, but not always, commercially viable products. Examples of real languages are COBOL, FORTRAN, PL/I, and (to a lesser extent) Algol and LISP. The properties that make a programming language a real language unfortunately also detract from the ease of verifying programs in that language. Most of these undesirable properties can be summed under the term "lack of semantic cleanliness."

The semantics of a programming language describe the meaning of statements in the language, expressed in some well-defined formal medium. A language has "clean" semantics if the definition of the language is elegantly expressible in some formal medium. There are many good reasons why real languages are not semantically clean:

. Most real languages have many operations. A real language incorporates the special interests of many groups of users, whose interests are not always compatible. Large numbers of features must often be added. These features not only complicate the semantics of the language, but often violate the spirit that motivated the initial conception of the language. PL/I is a good example of that. In a desire to overcome some of the difficulties of FORTRAN, COBOL, and Algol, the designers of PL/I created something larger than any of its ancestors. Considered alone, the size of real languages is a major obstacle to verification.

. Most real languages concede syntactic generality in the interest of an efficient implementation, in either the compiler or the generated code. Examples of these dependencies are limitations in the number of nestings (COBOL) or in the complexity of an arithmetic expression in certain syntactic positions (FORTRAN).

. Most real languages must have some features that deal with the hardware or operating system. The ENVIRONMENT DIVISION and Communication Module of COBOL are examples of these features. Standardization has served as a uniform interface between the language and the environment. However, the fact that a variable is SYNCHRONIZED or that there are 100 logical records in a physical record will not affect the correctness of a COBOL program, but may affect the performance of that program.

. Most real languages are the products of an evolving development, as illustrated by the fact that many real languages have numbers after their names to indicate the particular dialect in the sequence (COBOL 74, FORTRAN IV, Algol 60, LISP 1.5). In many cases, there is a desire for upward compatibility, so that bad features that could have been eliminated remain--"augmented" by the improvements.

Most of the important languages were created before the aesthetics of programming were well established. Thus, many real languages lack features such as strong typing, block structure, and flexible procedure and macro facilities. Structured programming practices are motivated by a desire to infuse these new aesthetics into the programming world. Perhaps verification will generate its own set of aesthetics to guide the design of future programming languages. Finally there is the problem that even if the semantics of a real language are clean, they are usually stated in natural language, e.g., in a standards manual.[12] A standards manual may suffice for programmers and language implementers, but it is not directly applicable to verification. Some attempts have been made to define language semantics formally (e.g., VDL[13] for PL/I). However, as long as there are no formal semantics for COBOL, it will be impossible to prove the correspondence between a language that is verified and a compiled version of such a language.

Before solutions to the problems of semantic cleanliness are considered, there is one major constraint to these solutions: the solutions must have minimum effect on the languages themselves. There is an understandable resistance by manufacturers to redesigning the programming languages that they support, and a similarly understandable resistance by users to recoding the software that they have written. Thus, the solution to the verification problem for real languages must be incremental. Research in new languages that support verification is very important, but the data processing community will ignore this research unless verification can be shown to be useful for currently existing languages.

The problem of language size has two aspects; syntactic and semantic. When a language has syntactic complexity (e.g., COBOL), there are many different ways to do the same thing. When a language has semantic complexity, there are many things that can be done. In cases where syntactic complexity exists, verification can be done on a program written in an internal form that is syntactically simple--there is only one way to do any given thing. Automatic translation from the external form to the internal form is relatively straightforward. Semantic complexity is handled primarily by subsetting, which entails choosing a sublanguage that includes only the desired semantic features. Some language constructs are useful (and even necessary), but can easily be misused.

This is precisely the problem with the GO TO. The solution to this type of problem takes several forms:

(1) Change the language.

(2) Establish management techniques to prevent abuse of the construct.

(3) Develop a preprocessor for the language that will allow desirable constructs in place of harmful ones.

For more information concerning these alternatives as applied to COBOL, see Section VII on structured programming and COBOL.

The semantics of a language can be specified by formulating an abstract machine whose instruction set is the set of commands in the programming language. Thus, a formal description of a language is the definition of such a machine. We define a machine that executes a subset of COBOL programs by means of a set of rules for generating verification conditions for the programs. Assuming that such a machine is actually consistent with a given COBOL compiler (something that may be difficult to determine), a program that is proved correct--using this verification condition gnerator--will actually run correctly when compiled by the given compiler. We use informal arguments to show that our formal definition of a COBOL subset is consistent with the ANSI standard.[12]

With regard to the features of a real language that are dependent on the hardware or the operating system, there are two strategies: to exclude them or to axiomatize them. Statements in COBOL's ENVIRONMENT DIVISION and items such as SYNCHRONIZED or the number of logical records per block can be excluded since they do not affect the outcome of the program. Special kinds of file input/output and communication with the operating system can be axiomatized as properties of the abstract machine that defines the programming language.

Several kinds of program properties are particularly important for real languages. There has been very little research done to date in the statement and proof of these kinds of properties. They are:

(1) Finite machine arithmetic

(2) Clean termination and run-time errors

(3) Validity of input data.

The issue of finite machine arithmetic is particularly acute in COBOL because data items have no more digits than they need for internal storage, while other languages have the (relatively large) word size of the machine. Thus, overflow and truncation occur often enough to be of concern. We consider these items in Section VI.

Clean termination was described earlier in this section. Because of the limited scope of this project, we did not deal with this issue in this report. Clean termination assumes the absence of run-time errors. However, such assumptions cannot safely be made, as is the case in hardware and operating system errors and in situations where input data is invalid. Run-time errors should be considered in efforts to verify programs in real language.

In verification, input data is assumed to be valid (with respect to type and range of values). One of the greatest difficulties in guaranteeing the reliability of programs in real languages is that such assumptions cannot be made. In other words, input data items are frequently faulty, and programs must be written to account for such situations. A real program will typically have several degraded modes of performance (without aborting the program), depending on the severity of the error. For example, even if a single record is destroyed, all other records may be processed correctly. There is a need in program verification to anticipate such occurrences and to make the input assertions for these programs as weak as possible.

E. Brief Discussion of the COBOL Language

COBOL is an extremely complex language--both syntactically and semantically. Since we could only apply verification techniques to a small subset of the entire language, we had to choose the issues (and parts of the language) that represent the most important aspects of COBOL. After briefly describing some characteristics of the COBOL language, we outline the coverage of features in the COBOL subset chosen by us relative to the modularization of ANS COBOL.

COBOL has a rich set of data types and operations. The area of most immediate concern for verification is called the "elementary data item." All computation in COBOL is character-oriented. Even numeric data items are treated as character strings with respect to assignment, truncation, and editing. Its control statements are also interesting, if not elegant to formalize. Elementary data items form the leaves in a tree-structured data declaration, of which the nonleaf nodes are called "group data items." A record is an entire tree of data declarations.

Input/output is very important in COBOL. Many programs follow the scheme:

```
Open files, initialize;
LOOP: Read input at end go to CLEANUP;
Process data;
Write output;
go to LOOP;
CLEANUP: Close files, etc.;
```

Thus, no useful COBOL program can be proved correct without some axiomatization of input/output.

Full ANS COBOL[12] is one of the most complicated programming languages, containing features that range from strings to interprocess communication. Thus, the subset of COBOL that we are verifying is small relative to the entire language, although the subset is a powerful language in itself. The subset provides arithmetic and relational operations on COBOL numeric data items, generalized control structures, and sequential input/output. A diagram of the components of the entire ANS COBOL language is shown in Figure I-2. In Figure I-2 , we indicate the parts of the language handled by the current verification system. The language is divided into twelve <u>modules</u>, which group related sets of features, and three <u>levels</u>, which are successively more sophisticated subsets of the modules.

A brief description of the contents of each module follows:

(1) <u>Nucleus</u>--Basic language constructs: control structures, data items.

(2) <u>Table Handling</u>--Arrays and subscripting.

(3) <u>Sequential I/O</u>--Reads and writes sequential files; a sequential file is a file whose records must be read in the order that they were written.

(4) <u>Relative I/O</u>--Reads from and writes to files whose records can be accessed in either sequential or random order, via a unique key that specifies a record's ordinal position within the file.

	LEVEL 1	LEVEL 2	LEVEL 3
NUCLEUS	▨	▨ ?	NULL
TABLE HANDLING	▨	▨ ?	NULL
SEQUENTIAL I/O	▨	▨ ?	NULL
RELATIVE I/O	NULL	?	?
INDEXED I/O	NULL	?	?
SORT-MERGE	NULL	?	?
REPORT WRITER	NULL	●	NULL
SEGMENTATION	NULL	●	●
LIBRARY	NULL	?	?
DEBUG	NULL	●	●
INTER-PROGRAM COMMUNICATION	NULL	?	?
COMMUNICATION	NULL	●	●

NULL - Nonexistent in ANS COBOL

▨ - Covered by Current Work

? - May be Covered in Future Work

● - No Immediate Plans for Coverage in Verification System

SA-3967-12

FIGURE I-2 STRUCTURE OF THE COBOL LANGUAGE

(5) Indexed I/O--Like relative I/O, except there can be multiple keys per record and a user can choose an arbitrary key when the record is written.

(6) Sort-Merge--Sorts a file according to fields within its records, or merges two or more identically sorted files of similar record structure.

(7) Report Writer--Allows the programmer to generate a file consisting of report (lineprinter) records solely by specifying the format of the report, rather than the algorithm for generating the report.

(8) Segmentation--Allows the programmer to specify the division of a COBOL program into segments whose object code can be overlayed in memory.

(9) Library--Enables the copying of sections of source code from centralized libraries.

(10) Debug--Allows the insertion of special sections of code for debugging, the execution of which can be switched on and off by compile- or object-time switches.

(11) Interprogram Communication--External procedure call and data-sharing mechanism.

(12) Communication--Message and synchronization facilities for concurrently executing programs.

The levels are numbered from 1 to 3, in order of increasing complexity. Most modules have no features at Level 1, as Figure I-2 indicates.

As shown in Figure I-2, our subset of COBOL provides most of the features of Level 1 and some features of Level 2, in the Nucleus, Table Handling, and Sequential I/O Modules. Most of the features left out (e.g., ALTER) are those that we consider undesirable because they substantially increase the difficulty of verification. Future work should expand the subset to include most of Level 2 in the aforementioned modules, and perhaps Level 2 of modules such as Relative and Indexed I/O. We view modules such as Library and Interprogram Communication as also being straightforward to verify, and modules such as Segmentation as not greatly affecting a program's input/output behavior. However, we view the Debug, Sort-Merge, and Communication Modules as being extremely difficult to verify, and as being of secondary importance at present.

There is a tendency to structure large COBOL programs around a large data base that has a unified set of data declarations that can be used by many programs. This is the aim of the CODASYL report on data bases,[14] in which the global data declarations for the whole system are called "schemas" and the local declarations for individual programs are called "subschemas." Although we have not looked at this report in depth, we find that the CODASYL report allows almost the same data declarations as COBOL 74 and permits almost arbitrary programs in COBOL 74 to operate on the shared data. Thus, we can use the same subsetting restrictions in the declarations and programs for the data base environment as we use for regular COBOL programs. We think that the problem of COBOL programs in cooperation can be better approached if the programs are forced to have consistent data declarations. Thus, the CODASYL work can by itself improve the reliability of large COBOL systems and be consistent with our effort in formally verifying COBOL programs.

II ANALYSIS OF COBOL WITH RESPECT TO VERIFICATION

A. Introduction

In this section we present a general analysis of COBOL for applicability of verification techniques. We try to identify some important general issues in COBOL that may have an impact on verification to be prepared for specific work, such as choosing a subset (Section IV) or designing the verification system (Section III). This discussion is different in that most literature on COBOL does not address formal verification.

If one considers a COBOL program to be running on a formally defined abstract machine that runs only COBOL programs, then one can try to formally define that abstract machine. If one encounters language constructs that are difficult to formally define and that lead to a needlessly complex definition, programs written in such a language will be difficult to verify. Often one must simplify the real COBOL language by subsetting it, so as to attain a tractable formal definition. Our analysis is based on such criteria. Inelegances in the formal definition can result (1) from machine-dependencies and representation issues that violate the level of abstraction intended to be provided by a COBOL machine, (2) from a multiplicity of special cases that must all be accounted for or (3) from constructs that are too general to allow powerful proof procedures to be applied.

Our analysis is based on two dimensions of the language--the data structures and the verbs. The interesting aspects of COBOL are characterized in these two dimensions. Data structures of COBOL are divided in these categories:

(1) Elementary data items with interesting properties
(2) Tree structured records and arrays of data declarations
(3) Files consisting of many records
(4) Multiple data declarations for the same storage areas.

Verbs of COBOL are divided into these categories:

(1) Assignment statements
(2) Control statements
(3) Input/output statements.

B. Elementary Data Items

Elementary (nonaggregate) data items can be either DISPLAY (a string of characters) or COMPUTATIONAL (a string of bits, e.g., machine integers and floating-point numbers). We decided to consider only DISPLAY data items--since COBOL's major application is to manipulate character-oriented data. DISPLAY items are characterized by a PICTURE specification (e.g., AAA, XXX, or S999V999), which is a format statement for representing the data item. A PICTURE specification implicitly declares the type of the data item to be one of the following: ALPHABETIC, NUMERIC, ALPHANUMERIC, NUMERIC EDITED, or ALPHANUMERIC EDITED. The EDITED data items are those whose values must be processed to be printed in a special format. The PICTURE specification also describes the size of the data item, its sign, and the position of its decimal point. The PICTURE specification relates to the possible values that a data item can assume so that a two-digit integer, for example, (PICTURE specification 99), has a maximum value of 99. In FORTRAN, on the other hand, a data item can only occupy the standard amount of storage space for an object of its type and takes on the same format each time it is printed.

Our major concern was with NUMERIC data items, for two reasons:

(1) These items are necessary for verifying nontrivial programs.

(2) Something is known about verifying programs in the numeric domain. (Domains such as strings have had little exploration.)

Originally, we had hoped to cover both NUMERIC and NUMERIC EDITED items, but the handling of NUMERIC EDITED items turned out to be too complicated to be handled in the allotted time.

Although a NUMERIC data item in COBOL is a character string, in most cases we consider the real numeric value that the data item represents. The only time we consider a NUMERIC data item's character string representation or PICTURE specification is in assignment statements and arithmetic operations, since truncation and overflow can occur there.

C. Tree-Structured Records

The "elementary items" are the leaves of the tree-structured record declarations of the COBOL DATA DIVISION. The nonleaf nodes of the tree are called "group data items." All data items in the tree have "level numbers" associated with them. For example, in the data declaration scheme

```
01  A
    02  B
    02  C
        03  D
    02  E
        03  F
        03  G,
```

B, D, F, and G are elementary items; A, C, and E are group data items; and 01, 02, and 03 are level numbers.

There are several implications of the tree-structure, all following from the fact that each data item has a context (i.e., its sequence of ancestors in the declaration tree). This allows lower-level data items to be referred to by naming an ancestor (e.g., in the MOVE and MOVE CORRESPONDING statements). It also allows two or more different data items to have the same name, so long as their contexts can be distinguished by qualification. For example, the data declaration scheme

```
01  A
    02  B
    02  D
01  C
    02  B
    02  E
```

allows references to the two different data items "B IN A" and "B IN C."

A data item can have as many qualifiers as are needed to ensure uniqueness. We handle qualification in the CSV (COBOL Subset for Verification).

Arrays can also be considered as group data items whose constituents are referenced in a special way. For example, an array A with 12 elements is declared as follows:

```
02  A  PICTURE 999  OCCURS  12  TIMES.
```

An array element is referenced as in FORTRAN (e.g., A(I)).

D. Files

Files are the most macroscopic data structures manipulated by COBOL programs. In fact, the behavior of a COBOL program can be described by stating properties of the input and output files manipulated by the program. Thus, files are not just another feature of the language, but essential elements. To deal with the semantics of COBOL, we must include files.

Files in COBOL are structures of records. Sequential files are a sequence of records. Writing a sequential file adds a record onto the sequence. To read a sequential file, a program starts by accessing the first record of the sequence. Subsequent rerd operations access the next record in the sequence, and so on. Direct access (relative) files and indexed sequential files are structures o files that can be referenced either sequentially (in an implicit way) or by key (in an explicit way). A record of a direct access file has one key that corresponds to its relative position within the file. A record of an indexed sequential file can have several keys that are independent of the record's relative location within the file.

We consider only sequential files in this phase of the project. These files are axiomatized as arrays of records. Each array has two variables: one indicates the length of the file and one poi[illegible] to the most recently accessed record. We foresee a straightforward axiomatization of relative and indexed files in the next phase of the project.

E. Multiple Definition of Storage Areas

In addition to providing a facility for the management of variables, COBOL also allows the programmer to use multiple definitions for the same areas of storage--similar to the FORTRAN COMMON or EQUIVALENCE statements. However, COBOL data items can be of arbitrary size, whereas FORTRAN data items come in fixed sizes related to the machine word. The biggest danger in multiple definition of storage areas is that a well-defined change to a data item defined in a certain way may cause an ill-defined (or possibly undefined) change to a data item that is defined differently but shares the same storage area. This possibility destroys the level of abstraction that is guaranteed by the concept of a data item in a higher level language. This level of abstraction is sometimes confused with the lower level abstraction of the machine's representation for the data item. Since higher level languages were created to avoid machine representations, overlapping data definitions circumvent a major purpose of a higher level language. However, overlapping data definitions improve the efficiency of prop-ams writteu in COBOL--even if they detract from their reliability.

COBOL provides three types of multiple data definition facilities: multiple records for a file, REDEFINES, and RENAMES. The first two facilities are similar. In both cases, a string of characters that belong to a data item (either group or elementary) can have another data definition. The primitive notion is that of a character. An example of REDEFINES is:

```
02   A.
     03   B   PICTURE   999.
     03   C   PICTURE   S999.
02   AA   REDEFINES A.
     03   BC  PICTURE   XXXXXX.
```

A graphical description of this data declaration occurs in Figure II-1. Note that assignment to B or C can change the value of BC, and vice versa. Note that any assignment to B or C will cause a valid assignment to BC, depending on the machine-dependent convention for representing the sign in C. However, there are many assignments to BC that would cause invalid values for either B or C. Multiple records per file are nothing more than a redefinition at the top level of the data declaration tree.

RENAMES is slightly different as shown in this example:

```
02   A.
     03   B   PICTURE   99.
     03   C   PICTURE    999.
02   D.
       03   E   PICTURE   9999.
66   X      RENAMES   B   THRU   E.
```

This declaration is depicted in Figure II-2. The construct allows a new group item to be defined, possibly to overlap other group items. However, the definitions of all elementary items are left intact. RENAMES preserves the level of abstraction provided by a COBOL elementary data item.

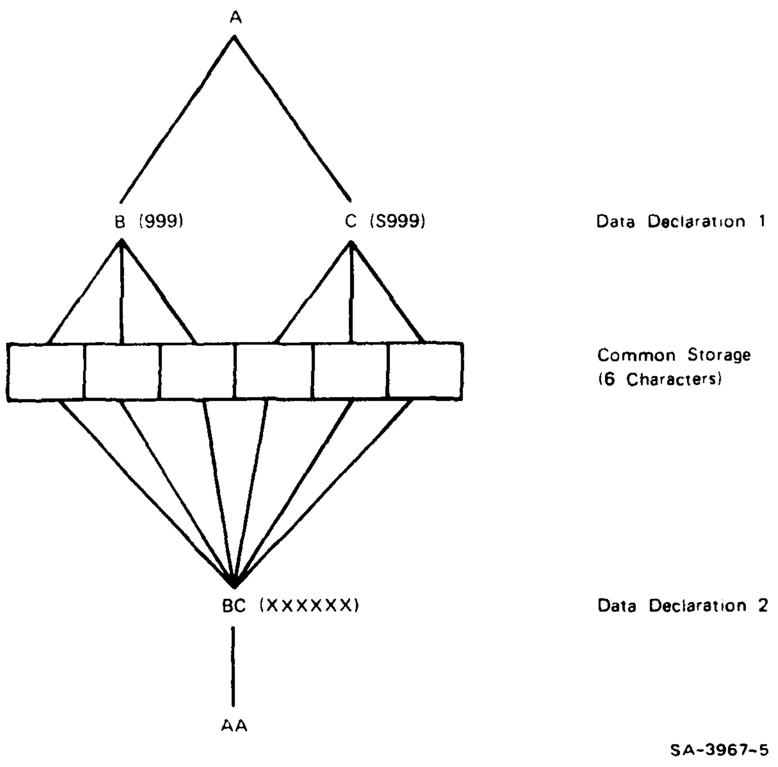


FIGURE II-1 MULTIPLE DEFINITION OF STORAGE BY THE REDEFINES STATEMENT

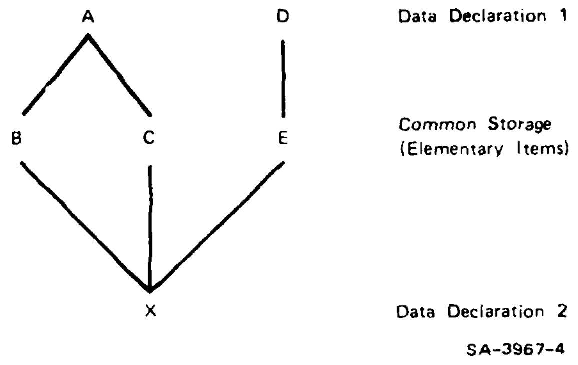


FIGURE II-2 MULTIPLE DEFINITION OF STORAGE BY THE RENAMES STATEMENT

We have left RENAMES, REDEFINES, and multiple-records-per-file out of the COBOL subset in this phase of the project. We envision including RENAMES in some future subset for verification, because it is a formally well-behaved construct. However, the other two constructs present difficult problems in the general case. Severe restrictions would be needed for redefinition at the character level. We would allow the REDEFINES statement when it does not take use of machine dependencies and underlying representation conventions.

F. Assignment and Computation Statements

These statements define new values for elementary data items in COBOL. Statements such as MOVE and MOVE CORRESPONDING are called "assignment statements," while statements such as ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE are called "computation statements." In Section II.B, we discussed how an elementary NUMERIC item possesses both a value and a PICTURE specification. Consider the statement

```
MOVE   A   TO   B.,
```

where A and B are elementary data items. In this type of operation, truncation and/or conversion may occur if the data items are of different types.

For now we have obviated the conversion problem by allowing only two types of data items--NUMERIC and ALPHANUMERIC--with no assignment from data of one type to data items of the other. However, in full COBOL this becomes a complex problem, with the addition of NUMERIC EDITED, ALPHANUMERIC EDITED, and ALPHABETIC type data items. The set of permissible conversions is described in Figure II-3. The permitted transfers marked by asterisks can violate the integrity of the receiving data item. These items should either be prohibited or validated by run-time type checking. There might be some difficulties, since COBOL does not perform this kind of checking until an error occurs. For example, use of a numeric data item in an arithmetic statement to which an alphanumeric data item has been moved, might not be possible.

Truncation is the deletion of trailing or leading characters because of an incompatibility of the PICTURE specifications of the sending and the receiving data items. With ALPHANUMERIC data items, the rightmost

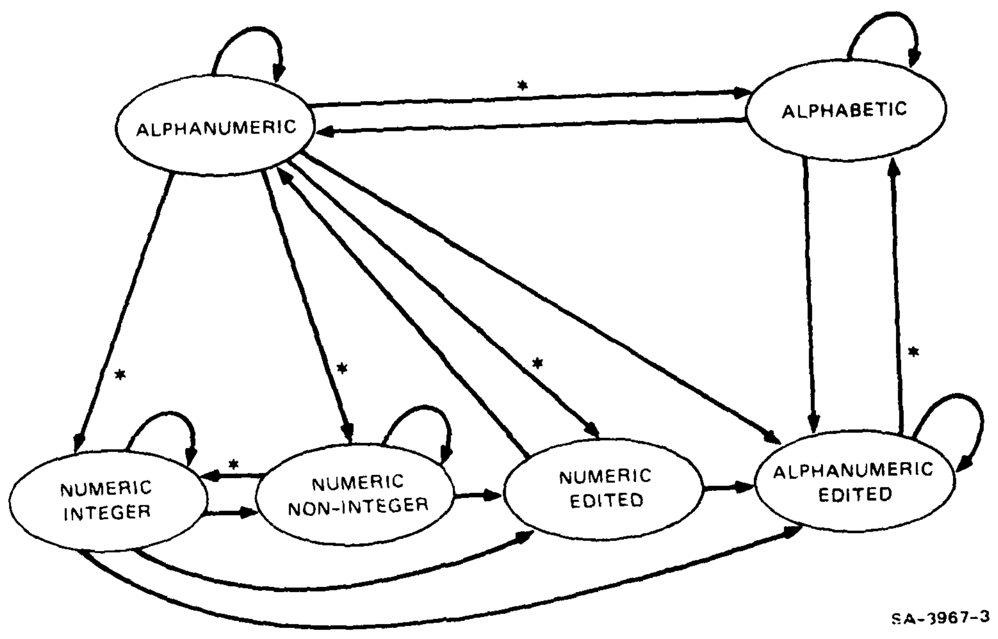


FIGURE II-3 DIAGRAM OF PERMISSIBLE MOVE OPERATIONS

characters are always truncated, unless the receiving item is declared as RIGHT JUSTIFIED. If the PICTURE specifications of an ALPHANUMERIC item indicate that it is too big to fit the sending item, spaces are filled in on the right (or left, when the item is RIGHT JUSTIFIED). In a MOVE statement among numeric data items, the decimal points are first aligned, then truncation or filling with zeros occurs at either end to fit the receiving data item.

We provide an assertion-language function called TRUNCATE (taking a PICTURE specification and a value as arguments) that describes this operation. However, there is some question about allowing the widespread use of truncation in COBOL programs. We believe that indiscriminate use of truncation is a major cause of unreliability in COBOL programs. In cases where truncation causes the loss of insignificant digits, this is a comparatively minor occurrence, although it may cause trouble. However, the truncation of significant digits of a data item, as the result of a MOVE statement, can cause serious problems. COBOL was designed to allow significant digits to be truncated in a MOVE statement. This is a convenient way to obtain the trailing digits in the destination item. However, it is difficult to tell whether that was intended or accidental, since both intentional and accidental uses have the same syntactic notation. If significant digit truncation were restricted to a statement such as

MOVE TRUNCATED A TO B.,

then the programmer's intention could be syntactically encoded in the statement.

Computation statements generate a SIZE ERROR when results produce overflow or truncation of significant digits (at run-time). This can be handled explicitly by the programmer by specifying a sequence of statements to be executed when a SIZE ERROR is detected. Otherwise a run-time error is generated.

The statement

MOVE CORRESPONDING <source> TO <destination>

takes two group data items as arguments. We define MOVE CORRESPONDING recursively. We determine the descendants of the source item and destination items, and the intersection set of their names. For each element of the intersection set whose source item is elementary, a MOVE is executed to the corresponding data item in the destination. Otherwise, a MOVE CORRESPONDING is performed from the source item to the corresponding destination item of the intersection set. This must be handled in the verification condition generator. The statement

MOVE <source> to <destination>

when used on group data items, indicates a moving of the contents of memory (without truncation or other processing) occupied by <source> to the memory area occupied by <destination>. This causes a loss of abstraction and is not allowed in the CSV.

G. Control Statements

Control statements in a programming language relate the lexical ordering of statements in a program to the dynamic ordering of execution of those program statements. A program is a fixed sequence of statements that defines the lexical ordering. However, when a program executes, the dynamic ordering of statements (the order of execution) depends on the input data. It is a function of the lexical ordering and of the data upon which the program operates. Several goals must be attained in choosing the set of control statements to be used in a programming language:

(1) Efficient description of any algorithm, in terms of both time and space.

(2) Minimum work for the programmer.

(3) Maximum simplicity and understandability of the control primitives themselves.

(4) Maximum understandability of the programs written using these control statements.

Goal (1) is satisfied by GO TO and a conditional statement, such as those contained in any assembler. This goal conflicts, to some extent, with all of the others. To satisfy Goal (2), language designers have introduced more complex control statements: e.g., looping constructs, procedures (i.e., call and return), case statements, coroutines, signals, switches, and the ALTER statement (unique to COBOL). Languages have resulted with a proliferation of primitives, some of which seem to be invented for a single special case. In regard to Goal (3), some work has been done in trying to find a minimum set of control primitives sufficient for most applications. Such efforts have also concentrated on eliminating the simplest but most primitive control statements (e.g., the GO TO), in attempting to satisfy Goal (4).

In terms of verification, Goals (3) and (4) are the most desirable. However, in terms of the COBOL subset, Goal (4) can be ignored, because it is always possible to write an incomprehensible (or difficult to verify) program using any given set of control primitives. Thus, adherence to Goal (4) depends largely on how the programs are written, not on the control primitives available. In any case, the subset should contain primitives that have an easily describable semantics.

At first glance, COBOL control statements seem to be simple, but they really contain much underlying complexity--making verification a potentially difficult task. The basic unit of execution is the statement or sentence. Sequences of statements are grouped together into paragraphs that are named. Control statements in COBOL take one of four schemes:

(1) Lexical ordering, either within or between paragraphs, which is the default.

(2) Unconditional transfer, via the GO TO. The object of a GO TO is a paragraph name. Control resumes at the first statement of the paragraph.

(3) Conditional execution, via the "IF a b ELSE c" statement. Either statement b or c is executed, depending on the value of conditional expression a.

(4) Procedural transfer of control (executing a sequence of statements and then returning to the point of transfer) via the PERFORM statement. The object of a PERFORM statement is either a paragraph name, or a pair of paragraph names (denoting the lexical sequence of paragraphs between the two names). All statements within the paragraph or sequence of paragraphs are PERFORMed. This construct can be used in conjunction with a condition or index variable to create loops.

Another aspect of control is the ALTER statement, which makes it possible to dynamically change the object of a GO TO statement. In programs with ALTER statements, the lexical structure becomes far removed from the dynamic structure, and programs with this property are very difficult to read. Since the ALTER statement does not do anything that cannot be accomplished by flags and conditional statements, we immediately remove it from consideration in the COBOL subset for verification.

All four types of control statements found in COBOL are also found in other programming languages, and present essentially no inherent problems. The difficulties occur in the way that these control statements interact. One problem is that loops and procedures are handled by the same syntactic mechanism, the PERFORM statement. The mechanism is different in languages such as FORTRAN and PL/I. Thus, the same paragraph can be invoked as a procedure or as a loop body. Procedures and loops are handled differently for verification. This issue is discussed in Section III. Another problem is that a paragraph that appears in a PERFORM statement can also be invoked by a GO TO or by its lexical order, further complicating verification. For example, the paragraph P may be invoked in any of the following situations:

```
PERFORM   P.                            (procedure call)
PERFORM   P VARYING I FROM 1 TO 10.
                                        (loop body)
GO TO P.                                (unconditional transfer)
Q.
   .
   .
   .
   MOVE X to Y.                         (lexical order)
P.
   .
   .
   .
```

Even though COBOL contains procedures, (a paragraph or sequence of paragraphs that are PERFORMed), there is no control statement that permits a direct return. Instead, control must pass to the last statement within the scope of the PERFORM. COBOL contains a nonexecutable statement that can be placed at the end of the scope of the PERFORM so that it can be the destination of a GO TO. This is the EXIT statement, and it must occur by itself in a paragraph. It resembles the CONTINUE statement of FORTRAN. Although the execution of procedures may be dynamically nested, COBOL provides no mechanism for a corresponding lexical nesting (such as the block structure of PL/I or Algol 60). All procedures in COBOL occur at the same lexical level of nesting. For example, consider the following scheme:

```
P1.
  .
  .
  .
  PERFORM P2.
  GO TO P3.

P2.
  .
  .
  .

P3.
  .
  .
  .
```

Paragraph P2 is at an inferior dynamic nesting to paragraphs P1 and P3, but has the same lexical nesting. This aspect of COBOL makes programs less readable.

To handle these complexities in COBOL control constructs, the verification condition generator must determine the manner in which a paragraph is invoked and must take appropriate action. There is some trade-off between the effort involved in verification condition generation and the length of the verification conditions that are produced. We discuss such issues in Section III.

H. Conclusions

The following features of COBOL present major problems in verification:

(1) Violation of the abstraction provided by COBOL

(2) Consideration of data items as strings

(3) Implementor-defined language features.

These problems are dealt with by excluding the offending features from the CSV. Strings will be handled in future work, but the other two features must be continually circumvented, either by exclusions from the subset or by showing that the offending features do not affect the program's input/output behavior.

Some other features that cause inelegancies in the proof process are dealt with in the current work:

(1) Semantically unclean control statements

(2) Finite machine arithmetic.

These issues are symptomatic of programming languages in general, and we intend--in future work--to find better ways to approach them.

III THE STRUCTURE OF THE COBOL VERIFICATION SYSTEM

A. Introduction

The parts of the verification system described in this section have been implemented using the INTERLISP language[15] on a PDP-10 running the TENEX operating system.[16] Although LISP is inefficient and its internal form is cumbersome to read, it is the easiest and most powerful of languages for writing programs that process structurally complex data. In future work, we intend to solve the problem of a cumbersome internal form using an infix printout routine to print out LISP expressions in the more natural infix form, without parentheses. However, we will use LISP, with its inefficiencies, for its advantages until a production system is built. The code and documentation for the modules of the system we have built are in the Appendix: the symbol table, the posttransduction processor, and the verification condition generator. We have borrowed the facilities for transduction grammars from other work at SRI.[17]

As described in Section I, to prove a program by using Floyd's method,[4] one must:

(1) Derive a set of mathematical formulas called "verification conditions" (VCs) whose validity is equivalent to the partial correctness of the program.

(2) Prove the validity of the VCs, either by hand or with the aid of a program called the deductive system.

Our approach to verification entails decomposing the process of producing verification conditions into three parts: syntax transduction, posttransduction processing, and low-level verification condition generation. Briefly, syntax transduction allows for the processing of a program in a syntactically complex (and possibly changing) language into a less syntactically complex internal form. Posttransduction processing translates the first internal form into a second internal form of reduced semantic complexity,so that the low-level verification condition generator can be as simple as possible. We have built all of these modules for COBOL verification condition generation. We believe that this approach to structuring the verification condition generator has substantially reduced the effort involved in writing the programs.

The deductive system would be decomposed into two parts: the heuristic deductive systems and the proof checker. Since the validity of a formula in first-order logic is undecidable (verification conditions are written in first-order logic), we need one or more heuristic systems that can attempt to arrive at proofs, based on strategies depending on particular high-level domains of inference (e.g., COBOL data structures). These heuristic deductive systems will often have human guidance. To assure logical soundness, the outputs of these heuristic deductive systems (i.e., the proofs) must be checked against a strict formal system. A program that checks a proof for logical soundness is called a "proof checker." The separation of heuristic deduction and proof checking results from the fact that we want the heuristic deductive systems to operate at a high level of abstraction--to take shortcuts and to use powerful rules of inference--but the proof checker must operate at the most primitive level of logical deduction. The separation means that only the proof checker need be correct, to guarantee valid deductions. That is, an incorrect proof caused by a bug in the heuristic deductive system will be revealed by the proof checker. We have built no machinery of this type specifically for COBOL verification, although other work at SRI[17] has been using this approach to make deductions about verification conditions for JOVIAL programs. We expect to use some of the components of the JOVIAL verification system in future work on COBOL.

The structure of the entire verification system is depicted in Figure III-1. Other parts of this section are devoted to the subsystems developed in this project: syntax transduction, posttransduction processing, and verification condition generation.

B. Syntax Analysis and Transduction

We use a table-driven language processor for initial processing of COBOL programs that are to be verified. Syntax transduction is the process of translating an input program from the standard form, in which COBOL programs are written by users of the language, to an abstract form with the same semantic properties but with a uniform structure easily manipulated by a posttransduction processor (the next phase of verification). The transduction phase is especially helpful in dealing with COBOL, which has extensive syntactic complexities that do not reflect comparable

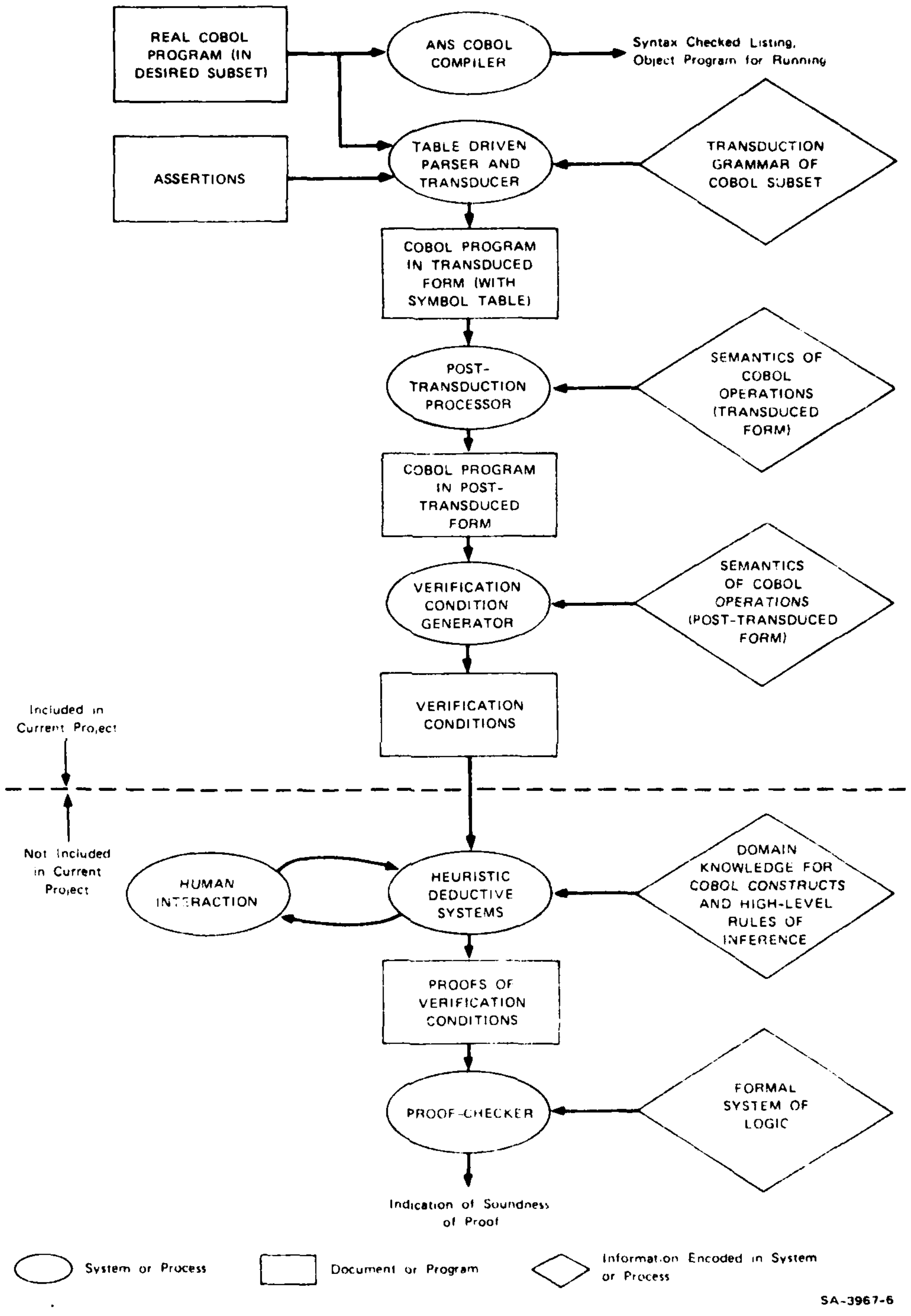

REAL COBOL PROGRAM (IN DESIRED SUBSET)
ANS COBOL COMPILER
Syntax Checked Listing, Object Program for Running
ASSERTIONS
TABLE DRIVEN PARSER AND TRANSDUCER
TRANSDUCTION GRAMMAR OF COBOL SUBSET
COBOL PROGRAM IN TRANSDUCED FORM (WITH SYMBOL TABLE)
POST-TRANSDUCTION PROCESSOR
SEMANTICS OF COBOL OPERATIONS (TRANSDUCED FORM)
COBOL PROGRAM IN POST-TRANSDUCED FORM
VERIFICATION CONDITION GENERATOR
SEMANTICS OF COBOL OPERATIONS (POST-TRANSDUCED FORM)
VERIFICATION CONDITIONS
Included in Current Project
Not Included in Current Project
HUMAN INTERACTION
HEURISTIC DEDUCTIVE SYSTEMS
DOMAIN KNOWLEDGE FOR COBOL CONSTRUCTS AND HIGH-LEVEL RULES OF INFERENCE
PROOFS OF VERIFICATION CONDITIONS
PROOF-CHECKER
FORMAL SYSTEM OF LOGIC
Indication of Soundness of Proof
System or Process
Document or Program
Information Encoded in System or Process
SA-3967-6

FIGURE III-1 STRUCTURE OF THE SRI COBOL VERIFICATION SYSTEM

semantic complexities. The point of the syntactic complexity of a language is to allow programmers to write in an expressive and natural format. While such a format is suitable for human consumption, it is inappropriate for the sorts of machine manipulation needed in verification. It is consequently beneficial to translate the external form to the syntactically much simpler abstract form that we have devised.

The correspondence between the internal and external forms is specified by a transduction grammar. Such a grammar consists of a set of BNF productions to describe the COBOL language, and a corresponding transduction for each production. A transduction is a LISP program that computes the abstract form of the language fragment specified by the associated production. Thus, we translate a COBOL program to an abstract form (called "Transduced COBOL") by using a parser to analyze a valid program into a "parse tree" according to the productions of the grammar, and then process the parse tree from bottom to top usirg transductions to obtain the parts of the desired Transduced COBOL program.

Our transduction grammar for COBOL (described in Section IV), together with various parsing and grammar manipulating tools, not only specifies the correspondence between COBOL and Transduced COBOL, but also constitutes an efficient algorithm for translating between the two languages. As a result of this translation, while a user may submit to the COBOL Verifier a general COBOL program (suitably annotated by logical assertions), parts of the system operating after transduction need to deal only with a very limited set of semantic primitives. For example, in the PROCEDURE DIVISION the translation expresses all ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE, and MOVE sentences (except for the CORRESPONDING option, which is handled separately) in terms of two semantic primitives SET$ and SETROUNDED$. The DATA DIVISION of a COBOL program is also transduced, but to a slightly different end. Instead of having a program as output, the transductions for the DATA DIVISION construct a symbol table from the tree-structured data declarations. This symbol table contains a data item's PICTURE specification, together with its ancestors and descendants in the declaration tree. This information is used in posttransduction processing and in verification condition generation for handling roundoff and truncation, for disambiguating qualified references to data items, and for interpreting commands like MOVE CORRESPONDING (see the general description of this verb in Section II and a specific semantic treatment in Section VI).

Finally, observe the advantage that derives from employing a COBOL Transduction Grammar (CTG) to drive the transducer. Although we have made a number of simplifying assumptions for the initial phase of the project, we can extend the subset of COBOL that is accepted simply by augmenting the CTG. Such extensions require no modification of the transducer.

As an example of part of a transduction grammar, consider the following example, not part of COBOL. For the COBOL transductions see Section IV. In this example, as in the verification system, the transduced form of the program is an S-expression in LISP.[15] The BNF rule has two alternatives as follows:

```
ifthenelse : : = IF boolexp statement ELSE statement.|
                 IF boolexp statement.,
```

where the upper-case words are terminal symbols and the lower-case words are nonterminal symbols. The transduction may look like this:

```
ifthenelse : : =  <IF$ T2 T3 T5>|
                  <IF$ T2 T3 NIL>
```

The angle brackets denote that the symbols between them are to be assembled into a list. IF$ is a special terminal symbol of the transduced form of the language. Tn (where n is a positive integer) denotes the transduction of the nth symbol in the corresponding BNF rule. Transduction for terminal symbols is an identity, while transduction for nonterminal symbols is governed by other transduction rules. Thus, T2 refers to the transduction of "boolexp" in both productions. NIL is the LISP atom referring to the null list. Thus, if we have the statement

```
IF x1 THEN x2 ELSE x3.,
```

its transduction is

```
(IF$ T(x1)  T(x2)  T(x3)),
```

where T(xn) is the transduction of xn (xn can contain complicated arithmetic or logical expressions).

C. Posttransduction Processing

A program in transduced COBOL looks much like a COBOL program: the statements have a one-to-one correspondence, and the control statements (and many of the verbs) are the same. Posttransduction processing reduces the semantic complexity by operations of the following types:

(1) Translating input and output statements into array accesses.

(2) Translating MOVE CORRESPONDING statements into MOVE with elementary data items.

(3) Translating PERFORM constructs into equivalent constructs containing assignments, tests, and branches.

(4) Adding machinery for qualification (unique naming) and truncation.

The result is an equivalent program that is written in a semantically much simpler language. The posttransduced program is longer than the program before transduction, however, and a certain trade-off is suggested: the verification system can be made more complex so as to handle programs in a semantically more complex language, but the intermediate forms (including the verification conditions) then will be more concise. The specific issues (relative to COBOL) in this trade-off will be discussed in Section VI.

The functions performed by posttransduction processing could have been performed in either the transduction phase or the verification condition generation phase. We wished to have an internal form (in Transduced COBOL) that resembled a real COBOL program, so we did not make the drastic program changes (involving control and verb changes) during the transduction phase. On the other hand, the design of programs to generate verification conditions can be an inordinately difficult task when done on a semantically complex language. Thus, we did not include posttransduction processing in the verification condition generator.

D. Verification Condition Generation

The output of posttransduction processing is a program that contains only the following kinds of statements:

(1) Assignment statements

(2) Array accesses

(3) Branches and sequencing

(4) Tests (IF _ _ ELSE _ statements).

A posttransduced COBOL program can be thought of as a simple flowchart scheme, with assertions at particular points in the flowchart graph and with assignment statements and array accesses in the flowchart boxes. A typical flowchart scheme, with numbered assertions, is depicted in Figure III-2. The verification condition generator must identify all simple paths in the program: those with an assertion at the beginning, some statements in the middle, and an assertion at the end. This involves some graph analysis, and yields results as shown in Figure III-3 when the graph of Figure III-2 is processed.

The next stage is to transform each of the simple paths into a verification condition. We use Hoare's axioms[18] to show how the four kinds of statements are handled. In Hoare's axioms, the construction P{S}Q means that if P holds before the execution of statement S, then Q holds after its execution. In the examples, we use a language containing only assignments, tests, and branches. In Section VI, we show the correspondence between COBOL and this simple language.

In the case of assignment, we have the following:

$$P\{X \leftarrow e\}Q \equiv P \supset Q^{X}_{e} \quad .$$

This means that the correctness of an assignment statement, with respect to assertions P and Q, is equivalent to the validity of the formula on the right-hand side, in which expression e has been substituted for the variable X. For example, this simple assignment to X yields the following VC that is trivially valid:

$$X < 0 \;\{X \leftarrow X + 1\}\; X < 1 \equiv$$
$$X < 0 \supset X + 1 < 1 \quad .$$

In the case of array accesses, we describe the following identities:

$$A(I) \equiv \text{SELECT}(A,\ I)$$
$$A(I) \leftarrow e \equiv \text{CHANGE}(A,\ I,\ e).$$

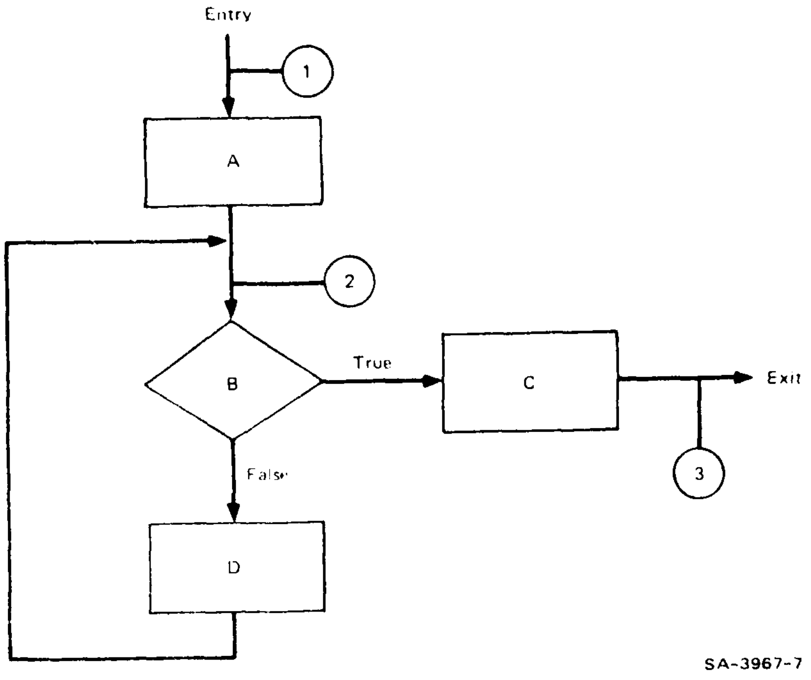


FIGURE III-2 A SIMPLE FLOWCHART SCHEMA

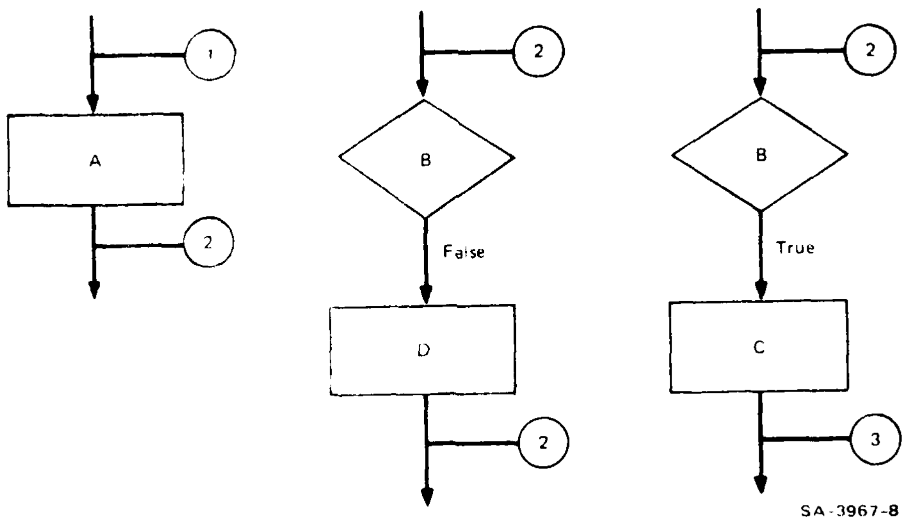


FIGURE III-3 THREE SIMPLE PATHS OF THE PROGRAM IN FIGURE III-2

The first identity applies only when A(I) occurs on the right-hand side of the assignment statement. We do not check the bounds of I here. The one transformation that applies is

$$P\{\text{CHANGE}(A,I,e)\}Q \equiv P \supset Q^{\text{SELECT}(A,x)}_{\underline{\text{if}}\ x = i\ \underline{\text{then}}\ e\ \underline{\text{else}}\ \text{SELECT}(A,x)} \ .$$

Here x is a quantified variable. An example of verification conditions in an array assignment is as follows:

$$(A[3] = 2 \wedge A[4] = 1)\{A[3] \leftarrow 5\}\ A[4] = 1 \equiv$$
$$(A[3] = 2 \wedge A[4] = 1) \supset (\underline{\text{if}}\ 3 = 1\ \underline{\text{then}}\ 5\ \underline{\text{else}}\ A[4]) = 1 \ .$$

Branches are handled by the path analyzer, but a single flowchart box might still have a sequence of statements inside it. The rule that defines verification condition for sequencing is as follows:

$$P\{S_1;\ S_2\}Q \equiv P\{S_1\}\ R_1 \wedge R_2\{S_2\}\ Q \wedge R_1 \supset R_2 \ .$$

In actual verification condition generation this rule is usually applied by "pushing" the consequent assertion (Q) through statement S_2 and then through S_1. This means that two different substitutions are performed in Q, the first yielding $R_1 (\equiv R_2)$, and the second yielding some predicate R_3, such that the path correctness is equivalent to $P \supset R_3$. The following is an example of a verification condition for simple sequencing:

$$X < 0\ \{X \leftarrow X + 1;\ X \leftarrow X + 2\}\ \ X < 3 \equiv$$
$$(X < 0\ \{X \rightarrow X + 1\}\ X < 1) \wedge (X < 1\ \{X \leftarrow X + 2\}\ X < 3) \wedge (X < 1 \supset X < 1)$$

The effects of branching are largely eliminated by a transduction to a flowchart scheme. However, this can be dealt with via the following Hoare axioms:

$$P_1\{\text{GO TO L}\} \wedge \dots \wedge P_n\{\text{GO TO L}\} \wedge Q\{L: \underline{\qquad\qquad}\} \equiv$$
$$(P_1 \vee \dots \vee P_n) \supset Q \equiv (P_1 \supset Q_1) \vee \dots \vee (P_n \supset Q_n) \ .$$

Since we are not concerned with the postconditions of the statements, they are omitted. In this axiom, there must be no more than n statements in the program that say "GO TO L". In COBOL, we require that assertions be placed at all labels that are the destinations of a GO TO, so that the assertions P_i do not have to be written. Instead $P_i \equiv Q_i$.

The effects of a test are as follows:

$P\{\text{test: } B(\underline{\text{true}})\}Q \equiv P \wedge B \supset Q$

$P\{\text{test: } B(\underline{\text{false}})\}\ Q \equiv P \wedge \neg B \supset Q$.

Two examples of verification conditions arising from a conditional statement are:

$X \geq 1\{\text{IF } X < 2 \text{ GO TO L1 ELSE GO TO L2}\} \wedge$

$X = 1\{\text{L1: } \_\_\_\_\_\_\}\quad \equiv$

$X \geq 1 \wedge X < 2 \supset X = 1$, and

$X \geq 1\ \{\text{IF } X < 2 \text{ GO TO L1 ELSE GO TO L2}\} \wedge$

$X \neq 1\{\text{L2:}\_\_\_\_\_\_\_\}\ \equiv$

$X \geq 1 \wedge \neg(X < 2) \supset X \neq 1$.

The verification conditions (logical formulas) for the program must then be proved valid for the program to be correct.

E. Conclusions

The method we used resulted in a simple system for generating COBOL verification conditions, so that most of the processing occurs in the phases of transduction and posttransduction processing.

IV SYNTAX OF THE COBOL SUBSET

A. Introduction

We have used the general guidelines presented in Section II to choose the syntax of the COBOL subset for verification (CSV). For each of the DATA DIVISION and the PROCEDURE DIVISION, we present a description of the features included in the subset and a discussion of its transduction grammar. Both the DATA DIVISION and the PROCEDURE DIVISION have features from the Nucleus, the Table Handling Module and the Sequential I/O Module. The features discussed are parts of Levels 1 or 2 in the COBOL language description. In the conclusion to this section, we discuss possible expansions to the subset.

B. DATA DIVISION Features

The DATA DIVISION contains the declarations for records and variables associated with files and with the program in general. Table IV-1* contains a list of the features of the DATA DIVISION, and an indication about whether or not they are allowed in the CSV. Table IV-2 contains a syntactic description of the DATA DIVISION of the CSV, in the same style as the ANSI COBOL manual. This enables a straightforward comparison of the CSV syntax with the syntax of COBOL 74 as described in [11]. The features of the Nucleus deal with declarations for program variables and records. The facilities used for declaring such variables and records (e.g., PICTURE) are also used in the declarations of variables and records associated with sequential files. We enumerate the constructs of the Nucleus first. We also indicate whether or not a feature is allowed in the CSV. If a feature is disallowed, we explain why.

WORKING-STORAGE SECTION (allowed). This is the section of the program that allows for program variables and records that are not associated with a particular file.

77-items (or Noncontiguous Working Storage) (allowed). These are individual variables (not records) in the WORKING-STORAGE SECTION.

*The Tables are at the end of the section.

Data-names or FILLER (allowed). Data-names can be either elementary or group data items in the declaration tree. FILLER denotes an unnamed data item.

JUSTIFIED (disallowed). For an alphabetic or alphanumeric data item, specifies whether right justification (filling with blanks on the left) is performed when a smaller data item is MOVEd to it. Left justification is the default. This was disallowed because we have not allowed MOVEs between alphabetic and alphanumeric data items of different sizes in the CSV.

Level-numbers (allowed). The ordering among level numbers describes the tree-structure of the data declarations.

PICTURE (partially allowed). PICTURE specifications enable a description of the precision and printing information for numeric data items, and of the size and printing information for nonnumeric data items. The PICTURE specifications that describe special printing instructions define edited data items, which we do not allow. The rules of editing and the assertions needed to describe an edited data item satisfactorily were deemed to be too complex to attempt at this time. The assertions that we prove in this initial effort deal with the values of data items and not with their printed forms. External form and an adequate formal treatment of string data is a subject for future research.

REDEFINES (disallowed). As described in Section II, we do not allow the REDEFINES concept because it allows the same area of storage to be described in two different ways, sometimes violating the abstraction provided by COBOL, which is based on the elementary data item. We acknowledge that this construct is a very powerful programming tool, and believe that some restriction of REDEFINES might provide much of the power without adversely affecting the abstraction. For example, a possible A REDEFINES B might be allowed only if both

(1) B is an elementary data item of form 9(n), A(n), or x(n).

(2) A is a group item, all of whose elementary items are of the same type as B. If B is numeric, then no elementary items of A may have a sign symbol S, but may have the virtual decimal point symbol V or precision symbols P. If B

is numeric, A may be an elementary numeric item having a virtual decimal point V or precision symbols P.

Thus, a possible data definition could be:

```
02 B PICTURE 9(9).
02 A REDEFINES B.
   03 A1 PICTURE 999 PPP.
   03 A2 PICTURE 9999V99   .
```

RENAMES (disallowed). RENAMES allows a sequence of contiguously defined elementary data items (possibly having different ancestors in the declaration tree) to be referred to by a single group data item. This allows multiple groupings of elementary data items. It has minor benefits in the MOVE and MOVE CORRESPONDING among group data items. We have disallowed the simple MOVE among group data items, so that the current benefits of this construct seem small at best. Ultimately we should be able to incorporate the construct with no great difficulty.

SIGN (disallowed). This feature allows a specification of the internal representation of the sign of a data item as being leading or trailing, and whether the sign is a separate character. Since we are not considering the internal representation of a data item--only its value--this feature is of no use in the current subset. Use of this feature does not affect the correctness of a COBOL program.

SYNCHRONIZED (disallowed). This feature allows the programmer to specify that a data item is aligned (either to the right or left) on a machine word boundary. This feature affects only the performance, and not the correctness, of a COBOL program: it deals only with a representation issue.

USAGE (disallowed). This feature allows the programmer to specify whether a data item is DISPLAY (character-oriented) or COMPUTATIONAL-n (stored in some format useful to the machine, e.g., binary integer or floating-point number). This again is a representation issue.

VALUE (disallowed). The initial value of a data item can be specified using this feature. There is no loss of generality by omitting this feature: a programmer can initialize a data item via an assignment statement at the beginning of the program.

The Table Handling Module has only a single feature in the DATA DIVISION: OCCURS (partially allowed). This feature allows the declaration of a data item to indicate an array. The array cannot be of variable length.

However, it will be straightforward to incorporate the use of variable-length arrays into a future subset.

The features of the DATA DIVISION in the Sequential I/O Module permit the declaration of files and their associated records. The FILE SECTION (allowed) contains zero or more file descriptions (allowed). Each file description contains one or more record descriptions (allowed), the components of which are described below. In the CSV, only one record description is allowed per file description.

BLOCK (disallowed). This optional feature declares how many logical records or characters are associated with a particular block (physical record). The verification system considers logical records only. Use of this feature does not affect the input/output behavior of the program, only its efficiency.

RECORD (disallowed). This optional feature described how many characters a record occupies. This clause is unnecessary even in full COBOL 74 (it is placed there for redundancy only), so that it can easily be done away with.

LABEL (disallowed). This feature (required in COBOL 74) allows the declaration of label records for a file as being either standard (according to the operating system) or omitted. It is disallowed for the same reason as the BLOCK clause. VALUE OF (disallowed) is a feature that either checks or sets a part of the label record.

DATA (disallowed). This optional feature specifies the data records associated with a file. Since the data records of a file are declared in the file description, this feature is unnecessary.

LINAGE (disallowed). This optional clause provides a system for keeping track of pages and lines within a page in a sequential file. It is useful in the generation of reports, and is omitted from the CSV because it is of small importance.

CODE-SET (disallowed). This optional clause specifies the character-code (e.g., EBCDIC or ASCII) used in the external representation of a file and is disallowed because the choice of character code does not affect the input/output behavior of a program.

C. Transduction Grammar for the DATA DIVISION

The transduction grammar for the DATA DIVISION yields an internal form (part of Transduced COBOL), but the internal form is not used in the verification of a COBOL program. However, the transduction grammar also creates a symbol table containing information on each data item. This symbol table is used by the transductions for the PROCEDURE DIVISION, for posttransduction processing, and for verification condition generation.

The symbol table contains information on files and on data items (both group and elementary). An entry for a file name contains its type (= FILE) and its corresponding record. An entry for a data item contains its type (= DATA ITEM) and either:

(1) If the item is a group data item, its ancestors in the declaration tree (back to the root), its immediate descendants, and the number of elements (if an OCCURS clause exists).

(2) If the item is an elementary item, its ancestors in the declaration tree (back to the root) and its PICTURE specification.

The ancestors (to the root) are used in qualification, and the descendants are used in evaluating the MOVE CORRESPONDING verb.

The major problem in constructing the symbol table is to take a flat description of the declaration tree and to make a tree structure out of it. For example, a COBOL program may contain a data declaration like this:

```
01 A.
   02 B.
      03 C PICTURE 99V99.
      03 D PICTURE XXX
   02 E. PICTURE 999.
   02 F.
      03 G PICTURE S999.
```

Although there may be indenting within the tree-structure, spaces are ignored in parsing, so that the only way to determine the tree-structure is by means of the level numbers. The transduction grammar makes each item's declaration into a list. If the item is a group item, then the group item will be the first element of a list with the descendants forming another list. The transduced version is as follows:

```
(((1 A)
  (((2 B)
    ((3 C 99V99)
     (3 D XXX)))
    (2 E 999)
    ((2 F)
     ((3 G S999))))))
```

Here is a graphic representation of the symbol table:

SYMBOL	LEVEL NUMBER	ANCESTOR PATH	DESCENDANTS	PICTURE
A	1		(B E F)	
B	2	(A)	(C D)	
C	3	(A B)		99V99
D	3	(A B)		XXX
E	2	(A)		999
F	2	(A E)	(G)	
G	3	(A E F)		S999

The transduction grammar for the DATA DIVISION of the CSV is shown in Table IV-3.

D. PROCEDURE DIVISION Features

The PROCEDURE DIVISION contains the actual code and assertions from which the verification conditions are generated. Table IV-4 contains a list of the features of the PROCEDURE DIVISION, and an indication about whether or not they are allowed in the CSV. Table IV-5 contains a syntactic description of the PROCEDURE DIVISION of the CSV, in the same style as the ANSI COBOL manual. We enumerate the constructs of the PROCEDURE DIVISION (in the Nucleus, Table Handling, and Sequential I/O Modules), describing the features and, if excluded from the CSV, the reasons for exclusion.

ACCEPT, DISPLAY (disallowed). The ACCEPT command allows for input from a console, or of the day, date, or time. It is disallowed because console can be simulated by the contents of a sequential file. Even if this construct were allowed, the commands issued from the console would have to be described as part of an array of records, whose properties are described by assertions. The DISPLAY command is disallowed for the same reasons.

ADD, SUBTRACT (allowed). All versions of this command are permitted, including ADD and SUBTRACT CORRESPONDING. All arithmetic statements allow rounding (as well as truncation, the default) and the SIZE ERROR clause.

ALTER (disallowed). This command can dynamically alter the flowchart of a program by changing the object of a GO TO statement. The set of possible paths through the program becomes too large to handle for verification. The effects of an ALTER statement can be simulated by the use of flags and conditional branches, which limit the number of possible program paths enough to permit verification.

COMPUTE (allowed). This is a generalized assignment to an arithmetic expression. The ROUNDED and SIZE ERROR options are allowed.

DIVIDE (partially allowed), MULTIPLY (allowed). We do not allow the REMAINDER option of the DIVIDE statement, but this option could be included in a future, expanded subset.

ENTER (disallowed). This verb permits the inclusion of statements in another language in a COBOL program, and is disallowed for obvious reasons.

EXIT (allowed). This is a no-op statement that allows exits from PERFORM blocks (somewhat like the FORTRAN CONTINUE) when the exit statement is contained in a paragraph at the end of a PERFORM block. This statement must be the only statement in a paragraph in which it appears.

GO (partially allowed). We do not allow the DEPENDING ON option, which could be included without adversely affecting verification. We also do not allow an option (to be used with the ALTER statement) in which a GO TO statement may have no arguments.

IF (allowed). This is the basic conditional statement.

INSPECT, STRING, UNSTRING (disallowed). We do not allow any string operators in this subset. We hope to include them in a future subset.

MOVE (partially allowed). We allow MOVE between elementary data items and MOVE CORRESPONDING, but we do not allow simple MOVEs between group data items (for reasons discussed in Section II). An issue that is closely related to the MOVE statement is that of type coercion. What happens if an item of Type A is moved into a variable of Type B? In some cases there is a simple answer, since the destination type subsumes the source type (e.g., INTEGER to REAL, ALPHABETIC to ALPHANUMERIC, NUMERIC to ALPHANUMERIC). In other cases, a policy of either automatic conversion or prohibition must be decided upon. In the case of moving a REAL to an INTEGER, this is solved by truncation. The remaining problems are ALPHANUMERIC to ALPHABETIC and ALPHANUMERIC to NUMERIC. The first case is not important because there are no operations on ALPHABETIC data items that can yield errors if the item has an ALPHANUMERIC value. The second case is not so simple: COBOL handles it by allowing a MOVE to be performed without checking, but by flagging an error if an operation is performed on the destination item. This is disasterous for verification, and it also seems harmful to good programming practice. We prefer some scheme that allows checking to be done when a MOVE is performed (perhaps optionally). We also favor an error category to be an optional part of the MOVE statement: "ON TYPE ERROR statement." There are many advantages to strongly typed languages, one of which is increased provability. These proposals are intended to make the type mechanism in COBOL stronger.

PERFORM (allowed). This is the basic textual abstraction and looping mechanism in COBOL.

STOP (partially allowed). We allow this statement without arguments only (an unconditional stoppage of execution). The STOP with arguments prints a message on the operator's console and permits restarting. The latter option would be difficult to axiomatize.

Some features in the Nucleus of the PROCEDURE DIVISION deal with expressions and data. We enumerate these features here. Qualification (allowed) enables the same name to be used for two (or more) different paragraphs or data items, when the ambiguity can be resolved by referring to a section name or to an ancestor in the declaration tree. This has added considerably to the complexity of the symbol table, and requires the verification system to make the names unique (at some time). The verification conditions of a program that contains many duplicated names can become extremely long. Clearly the ability to name two things with the same name is desirable (providing for such features as MOVE CORRESPONDING, for example). It remains to be seen what restrictions are necessary to allow shorter verification conditions.

Arbitrary arithmetic expressions are allowed in the CSV. However, we do not allow arbitrary conditions. Relation conditions (partially allowed) deal with the arithmetic relations $>$, $\geq$, $<$ $\leq$, $=$, and $\neq$. We allow arbitrary relations among numeric data items, but allow only $=$ and $\neq$ among nonnumeric data items. Class conditions (disallowed) state whether a data item is alphabetic or numeric, and can easily be incorporated into the CSV in future work. However, inclusion of this feature is closely related to the issue of type coercion (described in the paragraph on the MOVE statement). Condition-names and sign conditions (disallowed) could be included in a future subset, but there is no loss of generality from excluding them. Switch-status conditions (disallowed) depend on an implementor-defined switch and should not be allowed. Complex and combined conditions (allowed) are nothing more than the combining of simple conditions (those described above) with AND, OR, and NOT. Abbreviated combined conditions (disallowed) are a shorthand way of writing complex and combined conditions (e.g., " X > Y AND X > Z" translates to "X > Y AND Z"); they are needlessly difficult to process and also unnecessary.

In the PROCEDURE DIVISION of the Table Handling Module, there are two verbs SEARCH and SET (both disallowed), which deal with indexing variables (also disallowed). The only operation allowed on tables (or arrays) is the subscripting operation, in which a table is indexed by an expression, rather than a special indexing variable.

In the PROCEDURE DIVISION of the Sequential I/O Module there are primitives to manipulate sequential files.

CLOSE, OPEN (partially allowed). These statements are allowed, but without the REEL or UNIT designations that describe a file's implementation. The OPEN statement is not allowed with the I-O or EXTEND options, or with the REVERSED or NO REWIND designations. A file open for both INPUT and OUTPUT can be simulated (although not efficiently) by having two files--one for INPUT and one for OUTPUT.

READ, WRITE (partially allowed). The INTO option in READ and the FROM option in WRITE are disallowed (they involve a MOVE and then the READ or WRITE operations). The AT END clause in READ is permitted. All clauses in the WRITE statement dealing with pagination are disallowed.

REWRITE (disallowed). This operation deals with files that are open for both INPUT and OUTPUT, which is not allowed.

USE (disallowed). This statement allows the specification of procedures for input/output errors. The only error that we consider is end-of-file, which is handled with the AT END clause of READ.

E. Transduction Grammar for the PROCEDURE DIVISION

The transduced version of the PROCEDURE DIVISION is used in the generation of verification conditions. The transductions are usually a one-to-one translation of COBOL verbs except that

(1) Each verb has only one transduced syntax, subsuming all alternatives.

(2) All arithmetic assignment statements are reduced to the same internal form.

(3) Statements implying multiple operations are translated into multiple statements.

Other transformations to the internal form of the program are performed during posttransduction processing and verification condition generation.

Each COBOL sentence becomes a list. Each paragraph is a list whose first element is the keyword PARAGRAPH$, whose second element is the paragraph name, and whose other elements are its transduced sentences in order. Each section is a list whose first element is the keyword SECTION$, whose second element is the section name (if there are no sections a section name--FIRSTSECTION--is invented for the section consisting of all paragraphs), and whose other elements are the transduced paragraphs in order. To show the structure of the entire PROCEDURE DIVISION, we present the following simple COBOL program:

```
P1.    (ASSERT 1)
       MOVE 0 TO SUM.
       PERFORM P2 VARYING I FROM 1 BY 1
            UNTIL I GREATER THAN N
            (ASSERT 2).
       STOP RUN (ASSERT 3).
P2.    ADD A (I) TO SUM.
```

```
(PROCEDUREDIVISION$
 (SECTION$ FIRSTSECTION
   (PARAGRAPH$ P1
               (ASSERT 1)
               (SET$ SUM 0 NIL)
               (PERFORM VARYING
                        (DO$ P1 P1)
                        (I 1 1 (GT I N))
                        (ASSERT 2))
               (STOP (ASSERT 3)))
   (PARAGRAPH$ P2
               (SET$ SUM
                     (PLUS SUM
                           (SELECT A (I)))
                     NIL))))
```

To illustrate some interesting features of the transductions at the sentence level, we present some examples of COBOL verbs and their transductions. As an example of a simple one-sentence transduction, the sentence

```
CLOSE FILE1.
```

transduces to

```
(CLOSE FILE1).
```

The sentence

```
IF X GREATER THAN 0 NEXT SENTENCE ELSE GO TO P1.
```

transduces to

```
(IF (GT X 0)
    NEXT
    (GO P1))   ,
```

where NEXT can be interpreted by the verification condition generator. The PERFORM statement is an interesting case. The simple PERFORM,

```
PERFORM P1.,
```

transduces to

```
(PERFORM (ONCE$)
         (DO$ P1 P1)
         NIL NIL) .
```

ONCE$ is the option used to denote a single instance: the other alternatives are "n TIMES" and VARYING. DO$ indicates that the block of statements from P1 through P1 are the scope of the PERFORM (this can be expanded later). The two instances of NIL are places for the exit condition and the inductive assertion, when the statement is used as a loop. The PERFORM statement with a block of paragraphs,

```
    PERFORM P1 THRU PN.,
```

transduces to

```
     (PERFORM (ONCE$)
              (DO$ P1 PN)
           NIL NIL).
```

A simple COBOL loop looks like this:

```
    PERFORM P1
            VARYING I FROM 1 BY 1
            UNTIL I GREATER THAN N
            (ASSERT (P I)).
```

(the assertion is some predicate P on I), and its transduction looks like this:

```
     (PERFORM VARYING (DO$ P1 P1)
              (I 1 1 (GT I N))
              (ASSERT (P I))).
```

A nested PERFORM of the following form,

```
    PERFORM P1 VARYING I FROM 1 BY 1
              UNTIL I > N (ASSERT (P I))
         AFTER J FROM 1 BY 1
              UNTIL J > M (ASSERT (Q I J)).,
```

transduces to

```
     (PERFORM VARYING (PERFORM VARYING (DO$ P1 P1)
                                        (J 1 1 (GT J M))
                                        (ASSERT (Q I J)))
              (I 1 1 (GT I N))
              (ASSERT (P I))).
```

All simple arithmetic statements (not CORRESPONDING) transduce to SET$ (if truncated) and SETROUNDED$ (if ROUNDED). The COBOL sentence

```
    COMPUTE X = Y + Z.
```

transduces to

```
     (SET$ X (PLUS Y Z)
          NIL).
```

The NIL is where the SIZE ERROR clause would go if present. Notice that arithmetic and relational operators are translated to a single standard form for use in the verification system: PLUS, SUBTRACT, TIMES, DIVIDE, GT(>), LT(<), GTQ(≥), LTQ(≤), EQ, and NEQ. The arithmetic statements that have multiple results are transduced into multiple statements. For example, the statement,

```
    COMPUTE X1 ROUNDED, X2  = Y + Z; ON SIZE ERROR PERFORM P1.,
```

transduces to the pair of simple statements,

```
    (SETROUNDED$ X1 (PLUS Y Z)
                 (PERFORM (ONCE$)
                          (DO$ P1 P1)
                          NIL NIL))
    (SET$ X2 (PLUS Y Z)
          (PERFORM (ONCE$)
                   (DO$ P1 P1)
                   NIL NIL)).
```

All CORRESPONDING operations are separate, since they will be handled in posttransduction processing. The following sentence,

```
    ADD CORRESPONDING X TO Y.,
```

transduces to

```
    (ADDCORRESPONDING$ X Y NIL NIL).
```

F. Conclusions

The subset of COBOL that we have chosen for verification is small relative to the entire ANSI COBOL language,[12] yet it is a substantial programming language in itself--as complex as any for which verification has been attempted.

There are two important things in the Nucleus that are yet to be axiomatized:

- The handling of NUMERIC EDITED data items, with possible restrictions
- Character strings and their relation to numeric quantities.

Two unresolved issues affecting the ultimate choice of a subset are type coercion and a restriction of the REDEFINES construct.

In the Table Handling Module, the only major items left out of the CSV are indexing variables and the verbs that use them. It appears that they are easy to axiomatize, but we are unsure of their importance to COBOL programmers. All operations with indexing variables can be defined in terms of subscripting, so there is no loss of generality if we fail to incorporate them.

The major unresolved issue in the Sequential I/O Module deals with files that are open for simultaneous input and output. We intend to incorporate this into future COBOL subsets for verification.

It is not only important to axiomatize a large subset of a "real" language, but it is also important to be able to state and prove the important properties of the subset chosen. Thus, the choice of subset must be judged in terms of what can be proven, as well as its sheer size.

Table IV-1

SYNTAX OF THE DATA DIVISION OF THE CSV

GENERAL FORMAT FOR DATA DIVISION

```
DATA DIVISION.

[FILE SECTION.

[FD  file-name [record-description-entry] ... ] ...

[WORKING-STORAGE SECTION.

[77-level-description-entry ]     ]
[record-description-entry   ] ... ]
```

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

```
level-number {data-name-1}
             {FILLER     }

    [; {PICTURE}   IS character-string]
    [  {PIC    }                      ]

    [; OCCURS     integer-1 TIMES]
```

Table IV-2

FEATURES FOR THE DATA DIVISION OF THE CSV

DATA DIVISION Feature	COBOL 74		COBOL Subset for Verification	
	Level 1	Level 2	Level 1	Level 2
Nucleus				
WORKING-STORAGE SECTION	x*	†	x	
77-items	x		x	
Data name or FILLER	x		x	
JUSTIFIED	x			
Level number	x		x	
PICTURE	x		-‡	
REDEFINES	x			
RENAMES	x			
SIGN	x			
SYNCHRONIZED	x			
USAGE	x			
VALUE	x			
Table Handling				
OCCURS	x	x	x	
Sequential I/O				
FILE SECTION/file descriptions	x		x	
BLOCK contains	x	x		
RECORD contains	x			
LABEL records	x			
VALUE OF	x	x		
DATA RECORDS	x			
LINAGE		x		
CODE-SET	x			

* Feature included.
† (blank)--Feature nonexistent (in COBOL 74) or omitted (from COBOL subset for verification).
‡ Feature not totally included (in subset)

Table IV-3

TRANSDUCTION GRAMMAR FOR THE
DATA DIVISION OF THE CSV

```
*root*
#  datadivision
   (T1)

-----------------------------------

comma
#
   (NIL)
#  ,
   (T1)

-----------------------------------

datadescription
#  number dataname pictureclause occursclause .
   (<T1 T2 T3 T4>)

-----------------------------------

datadescriptions
#  datadescription
   (<T1>)
#  datadescription datadescriptions
   (<T1 ! T2>)

-----------------------------------

datadivision
#  DATA DIVISION . filesection workingstoragesection
   (<'DATADIVISION$ T4 T5>)

-----------------------------------

dataname
#  FILLER
   ('FILLER$)
#  symbol
   (T1)

-----------------------------------

filedescriptor
#  FD symbol . datadescriptions
   (<T1 (INSERTFILE T2 T4:1:1::1)
     (PROGN (INSERTRECORD T4:1:1:2 <T2>)
            (GETRECORD* T4))
     >)
```

```
--------------------------------------------

filedescriptors
#  filedescriptor
   (<T1>)
#  filedescriptor filedescriptors
   (<T1 ! T2>)

--------------------------------------------

filesection
#
   (NIL)
#  FILE SECTION . filedescriptors
   (<'FILESECTION$ ! T4>)

--------------------------------------------

is
#
   (NIL)
#  ARE
   (T1)
#  IS
   (T1)

--------------------------------------------

literal
#  number
   (T1)
#  string
   (T1)

--------------------------------------------

occursclause
#
   (NIL)
#  OCCURS number TIMES
   (T2)

--------------------------------------------

picture
#  PIC
   (T1)
#  PICTURE
   (T1)

--------------------------------------------
```

```
pictureclause
#
   (NIL)
#  semi picture is symbol
   (T4)
#  semi picture is number
   (T4)

-----------------------------------

recordlist
#  symbol
   (<T1>)
#  symbol comma recordlist
   (<T1 ! T3>)

-----------------------------------

semi
#
   (NIL)
#  ;
   (T1)

-----------------------------------

workingstoragesection
#
   (NIL)
#  WORKING-STORAGE SECTION . datadeclarations
   (<'WORKINGSTORAGESECTION$ ! T4>)

-----------------------------------
```

Table IV-4

SYNTAX OF THE PROCEDURE DIVISION OF THE CSV

GENERAL FORMAT FOR PROCEDURE DIVISION

FORMAT 1:

```
PROCEDURE DIVISION.
    {section-name SECTION.
    [paragraph-name.[sentence]... ]... }...
```

FORMAT 2:

```
PROCEDURE DIVISION.
    {paragraph-name. [sentence] ... } ...
```

GENERAL FORMAT FOR VERBS

```
ACCEPT identifier

ADD {identifier-1}  [, identifier-2]   ...   TO identifier-m [ROUNDED]
    {literal-1   }  [, literal-2   ]

    [, identifier-n  [ROUNDED] ...   [; ON SIZE ERROR imperative-statement]
ADD {identifier-1}  , {identifier-2}  [, identifier-3]  ...
    {literal-1   }    {literal-2   }  [, literal-3   ]

    GIVING identifier-m  [ROUNDED]  [, identifier-n  [ROUNDED] ]...

    [; ON SIZE ERROR imperative-statement]

ADD {CORRESPONDING}  identifier-1 TO identifier-2  [ROUNDED]
    {CORR         }

    [; ON SIZE ERROR imperative-statement]
```

```
CLOSE file-name-1 [, file-name-2] ...

COMPUTE identifier-1 [ROUNDED] [, identifier-2 [ROUNDED]] ...

    = arithmetic-expression [; ON SIZE ERROR imperative-statement ]

        {identifier-1}  [, identifier-2]
DISPLAY {literal-1   }  [, literal-2   ] ...

       {identifier-1}
DIVIDE {literal-1   } INTO identifier-2 [ROUNDED]

    [, identifier-3 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

       {identifier-1}      {identifier-2}
DIVIDE {literal-1   } INTO {literal-2   } GIVING identifer-3 [ ROUNDED]

    [, identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

       {identifier-1}    {identifier-2}
DIVIDE {literal-1   } BY {literal-2   } GIVING identifier-3 [ROUNDED]

    [, identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

EXIT

GO TO [procedure-name-1]

              {statement-1  } {; ELSE statement-2    }
IF condition; {NEXT SENTENCE} {; ELSE NEXT SENTENCE  }

     {identifier-1}
MOVE {literal     } TO identifier-2 [, identifier-3] ...

     {CORRESPONDING}
MOVE {CORR         } identifier-1 TO identifier-2

         {identifier-1}
MULTIPLY {literal-1   } BY identifier-2 [ROUNDED]

    [, identifier-3 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

         {identifier-1}    {identifier-2}
MULTIPLY {literal-1   } BY {literal-2   } GIVING identifier-3 [ROUNDED]

    [, identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

```
       (INPUT file-name-1  [, file-name-2] ...  )
OPEN   {OUTPUT file-name-3  [, file-name-4]...  }
       (I-O file-name-5  [, file-name-6] ...    )

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]
                           [{THRU   }                  ]

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]   {identifier-1}  TIMES
                           [{THRU   }                  ]   {integer-1   }

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]   UNTIL condition-.
                           [{THRU   }                  ]

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]
                           [{THRU   }                  ]

     VARYING  {identifier-2}   FROM  {identifier-3}
              {index-name-1}         {index-name-2}
                                     {literal-1   }

     BY  {identifier-4}        UNTIL  condition-1
         {literal-3   }

    [AFTER  {identifier-5}     FROM  {identifier-6}
    [       {index-name-3}           {index-name-4}
    [                                {literal-3   }


      BY  {identifier-7}       UNTIL  condition-2
          {literal-4   }
    [AFTER  {identifier-8}     FROM  {identifier-9}
    [       {index-name-5}           {index-name-6}
    [                                {literal-5   }

      BY  {identifier-10}      UNTIL condition-3]]
          {literal-6    }

STOP
```

```
SUBTRACT {identifier-1} [, identifier-2] ... FROM identifier-m [ROUNDED]
         {literal-1   } [, literal-2   ]

    [, identifier-n [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

SUBTRACT {identifier-1} [, identifier-2] ... FROM {identifier-m}
         {literal-1   } [, literal-2   ]          {literal-m   }

    GIVING identifier-n [ROUNDED] [, identifier-o [ROUNDED]] ...

    [; ON SIZE ERROR imperative-statement]

SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
         {CORR         }

    [; ON SIZE ERROR imperative-statement]

WRITE record-name
```

GENERAL FORMAT FOR CONDITIONS

```
RELATION CONDITION:
                              {IS [NOT] GREATER THAN}
                              {IS [NOT] LESS THAN   }
  {identifier-1             } {IS [NOT] EQUAL TO    } {identifier-2             }
  {literal-1                } {IS [NOT] >           } {literal-2                }
  {arithmetic-expression-1  } {IS [NOT] <           } {arithmetic-expression-2  }
  {index-name-1             } {IS [NOT] =           } {index-name-2             }

CLASS CONDITION:

    identifier IS [NOT] {NUMERIC   }
                        {ALPHABETIC}

NEGATED SIMPLE CONDITION:

    NOT simple-condition

COMBINED CONDITION:

    condition {{AND} condition} ...
              {{OR }          }
```

MISCELLANEOUS FORMATS

QUALIFICATION:

$$\left\{\begin{array}{l}\text{data-name-1}\\ \text{condition-name}\end{array}\right\} \quad \left[\left\{\begin{array}{c}\underline{\text{OF}}\\ \underline{\text{IN}}\end{array}\right\} \text{data-name-2}\right] \quad \ldots$$

$$\text{paragraph-name} \quad \left[\left\{\begin{array}{c}\underline{\text{OF}}\\ \underline{\text{IN}}\end{array}\right\} \text{section-name}\right]$$

SUBSCRIPTING:

$$\left\{\begin{array}{l}\text{data-name}\\ \text{condition-name}\end{array}\right\} \quad (\text{subscript-1} \quad [,\ \text{subscript-2} \quad [,\ \text{subscript-3}]]\ )$$

IDENTIFIER:

$$\text{data-name-1} \left[\left\{\begin{array}{c}\underline{\text{OF}}\\ \underline{\text{IN}}\end{array}\right\} \text{data-name-2}\right] \quad \ldots \quad \left[(\text{subscript-1} \quad [,\ \text{subscript-2} \quad [,\ \text{subscript-3}]\ )\right]$$

Table IV-5

FEATURES OF THE PROCEDURE DIVISION OF THE CSV

PROCEDURE DIVISION Feature	COBOL 74		COBOL Subset for Verification	
	Level 1	Level 2	Level 1	Level 2
Nucleus	--[‡‡]	--	--	--
Verbs	--	--	--	--
ACCEPT	x[*]	x	x	0[‡]
ADD	x	x	x	x
ALTER	x	x	0	0
COMPUTE	0	x	0	x
DISPLAY	x	x	x	
DIVIDE	x	x	x	-[†]
ENTER	x	0	0	--
EXIT	x	0	x	0
GO	x	x	-	0
IF	x	x	x	x
INSPECT	x	0	0	0
MOVE	x	x	x[§]	x
MULTIPLY	x	x	x	x
PERFORM	x	x	x	x
STOP	x	0	-	0
STRING	0	x	0	0
SUBTRACT	x	x	x	x
UNSTRING	0	x	0	0
Internal Constructs	--	--	--	--
Qualification	x	0	x	0
Arithmetic Expressions	0	x	0	x
Conditions	--	--	--	--
Relation conditions	x	x[**]	x	x
Class conditions	x	0	0	0
Condition-names	0	x	0	0
Switch-status	x	0	0	0
Sign conditions	x	0	0	0
Complex and combined conditions	0	x	0	x[††]
Abbreviated combined conditions	0	x	0	0
Size Error	x	0	x	0
Sequential I O	--	--	--	--
CLOSE	x	x	-	0
OPEN	x	x	x	0
READ	x	0	-	0
REWRITE	x	0	0	0
USE	x	x	0	0
WRITE	x	x	-	0
Table Handling	--	--	--	--
SEARCH	0	x	0	0
SET	x	0	0	0
Subscripting	x	0	x	0
Indexing	x	0	0	0

* Feature included.
** Nonnumeric equality only.
† Feature not totally included (in subset).
‡ Feature nonexistent (in COBOL 74) or omitted (from COBOL subset for verification).
§ No MOVE between group data items.
†† Complex conditions consisting only of simple conditions allowed in subset.
‡‡ Not applicable.

Table IV-6

TRANSDUCTION GRAMMAR FOR THE
PROCEDURE DIVISION OF THE CSV

```
proceduredivision
#  PROCEDURE DIVISION . paragraphs
   (<'PROCEDUREDIVISION$ <'SECTION$ 'FIRSTSECTION ! T4>>)
#  PROCEDURE DIVISION . sections
   (<'PROCEDUREDIVISION$ ! T4>)

-------------------------------------

at
#
   (NIL)
#  AT
   (NIL)

-------------------------------------

computetarget
#  computetarget1
   (<T1>)
#  identifier , computetarget
   (<< SET$ T1> ! T3>)
#  identifier ROUNDED , computetarget
   (<<'SETROUNDED$ T1> ! T4>)

-------------------------------------

computetarget1
#  identifier
   (<'SET$ T1>)
#  identifier ROUNDED
   (<'SETROUNDED$ T1>)

-------------------------------------

condition
#  condition OR condition2
   (<T2 T1 T3>)
#  condition2
   (T1)

-------------------------------------

condition2
#  condition2 AND condition3
   (<T2 T1 T3>)
#  condition3
   (T1)
```

```
-----------------------------------------

condition3
#  NOT condition3
   (<T1 T2>)
#  condition4
   (T1)

-----------------------------------------

condition4
#  ( condition )
   (T2)
#  simplecondition
   (T1)

-----------------------------------------

connector
#  BY
   (T1)
#  FROM
   (T1)
#  INTO
   (T1)
#  TO
   (T1)

-----------------------------------------

corresponding
#  CORR
   (NIL)
#  CORRESPONDING
   (NIL)

-----------------------------------------

corrop
#  ADD
   ('ADDCORRESPONDING$)
#  SUBTRACT
   ('SUBTRACTCORRESPONDING$)

-----------------------------------------

dividearguments
#  expression BY expression
   (<T1 T3>)
#  expression INTO expression
   (<T3 T1>)
```

```
-----------------------------------------

elseclause
#
   ('NEXT)
#  semi ELSE NEXT SENTENCE
   ('NEXT)
#  semi ELSE sentence
   (T3)

-----------------------------------------

endcondition
#
   (NIL)
#  ; at END sentence
   (T4)
#  at END sentence
   (T3)

-----------------------------------------

errorcondition
#
   (NIL)
#  ; on SIZE ERROR sentence
   (T5)

-----------------------------------------

expression
#  expression + expression2
   (<'PLUS T1 T3>)
#  expression2
   (T1)
#  expression - expression2
   (<'SUBTRACT T1 T3>)

-----------------------------------------

expression2
#  expression2 * expression3
   (<'TIMES T1 T3>)
#  expression2 / expression3
   (<'DIVIDE T1 T3>)
#  expression3
   (T1)

-----------------------------------------

expression3
#  expression3 ** expression4
   (<'EXP T1 T3>)
```

```
#  expression4
   (T1)

------------------------------------

expression4
#  ( expression )
   (T2)
#  + expression4
   (T2)
#  - expression4
   (<'MINUS T2>)
#  ZERO
   (0)
#  ZEROES
   (0)
#  ZEROS
   (0)
#  identifier
   (T1)
#  number
   (T1)
#  string
   (T1)

------------------------------------

expressions
#  expression
   (<T1>)
#  expression , expressions
   (<T1 ! T3>)

------------------------------------

filename
#  symbol
   (T1)

------------------------------------

filenames
#  filename
   (<T1>)
#  filename , filenames
   (<T1 ! T3>)
#  filename filenames
   (<T1 ! T2>)

------------------------------------

identifier
#  symbol subscriptlist qualifiers
```

```
   ((if T2 then <'SELECT
        (if T3 then <'QUAL T1 ! T3> else T1)
        T2> elseif T3 then <'QUAL T1 ! T3> else T1))
```

```
identifiers
#  identifier
   (<T1>)
#  identifier , identifiers
   (<T1 ! T3>)
```

```
indexname
#  symbol
   (T1)
```

```
iotype
#  INPUT
   ('OPENINPUT$)
#  OUTPUT
   ('OPENOUTPUT$)
```

```
is
#
   (NIL)
#  IS
   (NIL)
```

```
of
#  IN
   (NIL)
#  OF
   (NIL)
```

```
on
#
   (NIL)
#  ON
   (NIL)
```

```
operator
#  ADD
   ('PLUS)
#  DIVIDE
   ('DIVIDE)
#  MULTIPLY
   ('TIMES)
#  SUBTRACT
   ('SUBTRACT)

-----------------------------------

paragraph
#  paragraphname . sentences
   (<'PARAGRAPH$ T1 ! T3>)

-----------------------------------

paragraphname
#  symbol
   (T1)

-----------------------------------

paragraphs
#  paragraph
   (<T1>)
#  paragraph paragraphs
   (<T1 ! T2>)

-----------------------------------

performbody
#  procedurename
   (<'DO$ T1 T1>)
#  procedurename thru procedurename
   (<'DO$ T1 T3>)

-----------------------------------

performcontrol
#  UNTIL condition assertion
   (<T2 T1 T3>)
#  expression TIMES assertion
   (<T2 T1 T3>)
#  varying identifier FROM expression BY expression UNTIL condition
assertion
   (<T1 <T2 T4 T6 T8> T9>)

-----------------------------------

performcontrols
#
```

```
    (NIL)
#   performcontrol performcontrols
    (<T1 ! T2>)

---------------------------------------------

procedurename
#   symbol
    (T1)
#   symbol of symbol
    (<T1 T3>)

---------------------------------------------

procedurenames
#   procedurename
    (<T1>)
#   procedurename , procedurenames
    (<T1 ! T3>)

---------------------------------------------

qualifiers
#
    (NIL)
#   of symbol qualifiers
    (<T2 ! T3>)

---------------------------------------------

recordname
#   symbol
    (T1)

---------------------------------------------

relationoperator
#   NOT relationoperator2
    ((SELECTQ T2 ((QUOTE EQ)
                  'NEQ)
                 ((QUOTE NEQ)
                  'EQ)
                 ((QUOTE LT)
                  'GTQ)
                 ((QUOTE GTQ)
                  'LT)
                 ((QUOTE LTQ)
                  'GT)
                 ((QUOTE GT)
                  'LTQ)
                 (HELP '
        'Error in reduction of first alternative of relationoperator.")))
#   relationoperator2
```

```
   (T1)

----------------------------------------

relationoperator2
#  <
   ('LT)
#  =
   ('EQ)
#  >
   ('GT)
#  EQUAL to
   ('EQ)
#  GREATER than
   ('GT)
#  LESS than
   ('LT)

----------------------------------------

rounded
#
   (NIL)
#  ROUNDED
   (T1)

----------------------------------------

section
#  sectionname SECTION . paragraphs
   (<'SECTION$ T1 ! T4>)

----------------------------------------

sectionname
#  symbol
   (T1)

----------------------------------------

sections
#  section
   (<T1>)
#  section sections
   (<T1 ! T2>)

----------------------------------------

semi
#
   (NIL)
#  ;
   (NIL)
```

```
------------------------------------------

sentence
#  sentence1
   (T1)
#  sentence2
   ((if T1::1 then <'DO$ ! T1> else T1:1))

------------------------------------------

sentence1
#  CLOSE filenames
   (<T1 ! T2>)
#  EXIT
   (NIL)
#  GO to procedurename
   (<T1 T3>)
#  IF condition thenclause elseclause
   (<T1 T2 T3 T4>)
#  PERFORM performbody performcontrols
   ((if T3 tnen (for (X R_T2)
                     in
                     (REVERSE T3)
                     do R_ <'PERFORM X:1 R X:2 X:3> finally
                     (RETURN R))
          else <'PERFORM ' (ONCE$)
          T2 NIL NIL>))
#  READ filename endcondition
   (<T1 T2 T3>)
#  STOP RUN assertion
   (<T1 T3>)
#  WRITE recordname
   (<T1 T2>)
#  assertion
   (T1)
#  corrop corresponding identifier connector identifier rounded
errorcondition
   (<T1 T3 T5 T6 T7>)

------------------------------------------

sentence2
#  COMPUTE computetarget = expression errorcondition
   ((for X in T2 collect <! X T4 T5>))
#  GO to procedurenames DEPENDING on expression
   ((for I to (LENGTH T3)
         collect
         (<'IF <'EQ$ T6 I> <'GO (CAR (NTH T3 I))
           > 'NEXT >)))
#  OPEN iotype filenames
   ((for X in T3 collect <T2 X>))
#  MOVE expression TO identifiers
   ((for X in T4 collect <'SET$ X T2 NIL>))
```

```
#  MOVE corresponding identifier TO identifiers
   ((for X in T5 collect < MOVECORRESPONDING$ T3 X>))
#  ADD expressions GIVING computetarget errorcondition
   ((for X in T4 collect <! X <'PLUS ! T2> T5>))
#  ADD expressions TO computetarget errorcondition
   ((for X in T4 collect <! X <'PLUS X:2 ! T2> T5>))
#  SUBTRACT expressions FROM computetarget errorcondition
   ((for X in T4 collect <! X <'SUBTRACT X:2 <'PLUS ! T2>> T5>))
#  SUBTRACT expressions FROM expression GIVING computetarget
errorcondition
   ((for X in T6 collect <! X <'SUBTRACT T4 <'PLUS ! T2>> T7>))
#  MULTIPLY expression BY computetarget errorcondition
   ((for X in T4 collect <! X <'TIMES T2 X:2> T5>))
#  MULTIPLY expression BY expression GIVING computetarget errorcondition

   ((for X in T6 collect <! X <'TIMES T2 T4> T7>))
#  DIVIDE expression INTO computetarget errorcondition
   ((for X in T4 collect <! X <'DIVIDE X:2 T2> T5>))
#  DIVIDE expression INTO expression GIVING computetarget errorcondition

   ((for X in T6 collect <! X <'DIVIDE T4 T2> T7>))
#  DIVIDE expression BY expression GIVING computetarget errorcondition
   ((for X in T6 collect <! X <'DIVIDE T2 T4> T7>))

-------------------------------------

sentences
#  sentence1 .
   (<T1>)
#  sentence1 . sentences
   (<T1 ! T3>)
#  sentence2 .
   (T1)
#  sentence2 . sentences
   (<! T1 ! T3>)

-------------------------------------

simplecondition
#  expression is relationoperator expression
   (<T3 T1 T4>)

-------------------------------------

subscriptlist
#
   (NIL)
#  ( subscripts )
   (T2)

-------------------------------------
```

```
subscripts
#   expression
    (<T1>)
#   expression , subscripts
    (<T1 ! T3>)
```

```
than
#
    (NIL)
#   THAN
    (NIL)
```

```
thenclause
#   NEXT SENTENCE
    ('NEXT)
#   semi sentence
    (T2)
```

```
thru
#   THROUGH
    (NIL)
#   THRU
    (NIL)
```

```
to
#
    (NIL)
#   TO
    (NIL)
```

```
varying
#   AFTER
    ('VARYING)
#   VARYING
    ('VARYING)
```

V. THE ASSERTION LANGUAGE FOR COBOL VERIFICATION

A. General

The deductive system in a program verification system will attempt to prove the validity of a formula in first-order logic, the language of mathematical theorems. When one builds a formal mathematical system, one starts with axioms and attempts to prove theorems based on them. In program verification, the axioms are the semantics of the programming language, the program, and the assertion language. The theorem states that the program is correct with respect to the assertions.

Thus, the assertion language must be the language of mathematics (in this case first-order logic, integers, and real numbers) plus some constructs that apply directly to COBOL. We have used very few of the latter--only those that deal with arrays, truncation, rounding, and overflow in COBOL data items.

We believe that the inclusion of more language-oriented constructs in the assertion language will shorten the assertions, making them easier to read and write, and will also simplify proof. In the last section, we describe some ideas in that direction that have not been implemented.

B. Basic Assertion Language

There are three elements of the basic assertion language:

(1) First-order logic with equality

(2) Real and integer arithmetic operators and relations

(3) User-defined functions and predicates.

First-order logic contains the quantifiers $\forall$, $\exists$; the connectives $\wedge$, $\vee$, $\neg$, $\supset$; the equality symbol =; and the symbols, predicates, and functions used in the logical formula. The LISP function names for the quantifiers and connectives are, respectively: FORALL, EXISTS, AND, OR, NOT, IMPLIES, and EQ. Often basic set theory is useful in connection with first-order logic.

The operations on arithmetic items are simply +, -, *, /, and unary minus (and exponentiation, logarithms, modulo arithmetic). The relations

BEST AVAILABLE COPY

are simply =, ≠, >, ≥, ≤, and <. All axioms on these operations and relations apply (e.g., commutativity and transitivity). The LISP function symbols for the arithmetic and relational operators, respectively, are PLUS, SUBTRACT, TIMES, DIVIDE, MINUS, EQ, NEQ, GT, GTQ, LTQ, and LT.

Other constructs, such as conditional expressions (from Algol 60) are also used. An example of the use of a conditional expression is

$$x = \underline{\text{if}}\ b\ \underline{\text{then}}\ y\ \underline{\text{else}}\ z,$$

and this translates to the logical formula

$$(b \supset x = y) \wedge (\neg b \supset x = z).$$

A facility for the user to define functions, predicates, and syntactic constructs is also useful. In this context, all special constructs relating specifically to COBOL could be formally defined. Some strict laws of definition (such as those in Reference 26) should be used in creating new definitions, so that the soundness of any proofs based on the definitions is guaranteed.

The rules of inference are the basic ones for first-order logic (e.g., _modus ponens_). Other "rules" can be derived as theorems.

C. Special Functions for COBOL

The special functions that we have used in our assertion language are concerned with particular features of COBOL arrays and numeric data items. These functions are described in the following paragraphs.

The SELECT function for array access is briefly described in Section III. It has no definition, except that SELECT(A,I) returns the I th value of the array A. Its value is changed via an array assignment CHANGE(A, I, V), which changes the value of the I th element of the array A to V. Its formal semantics is described in terms of the following Hoare axiom [8]:

$$P\{\text{CHANGE}(A,I,e)\}Q \equiv$$

$$P \supset Q^{\text{SELECT}(A,x)}_{\underline{\text{if}}\ x = I\ \underline{\text{then}}\ e\ \underline{\text{else}}\ \text{SELECT}(A,x)} .$$

This means that when a single value of array A changes (the I^{th} value), a conditional substitution is made for the I^{th} value of the array only. However, for a given array access SELECT(A,x), where x is an arbitrary expression, it may not be known at verification time whether or not x is equal to I. Thus, the above conditional expression must be substituted into Q for every instance of SELECT(A,x). We can represent the size of the array A by the function LENGTH(A).

Two assertion-language functions are associated with assignments to numeric data items. In a normal assignment statement, truncation takes place so that the new value of the receiving data item "fits" its PICTURE specification. In an arithmetic statement with the ROUNDED option, rounding of the least significant digits takes place instead of truncation. In both operations, the most significant digits will be lost if the absolute value of the item is "too big" for its picture specification. We supply two functions, TRUNCATE and ROUND, to perform these operations; both functions take a value and a PICTURE specification as an argument, and return a new value. For example,

TRUNCATE(123.46, 999V9) = 123.4

ROUND(123.46, 999V9) = 123.5

TRUNCATE(-1234.56, S999V99) = -234.56

ROUND(-234.56, 999V99) = 234.56

We define ROUND and TRUNCATE in terms of primitive and intermediate-level constructs, and then define the intermediate-level constructs in terms of the primitives presented here:

ABS(v) = absolute value of v

DECIMALDIGITS(p) = number of digits to the right of the virtual decimal point of p

TOTALDIGITS(p) = total number of digits in p

MOD(v1,v2) = v1 mod v2 .

In this case, v is a value and p is a PICTURE specification. We define TRUNCATE and ROUND as follows:

```
TRUNCATE(v,p) =
     if ABS(v) > MAXVAL(p)
          then MAXVAL(p) * PSIGN(v,p)
        else (ABS(v) - EXTRADIGITS(v,p)) * PSIGN(v,p)
ROUND(v,p) =
     if ABS(v) > MAXVAL(p)
          then MAXVAL(p) * PSIGN(v,p)
       else if EXTRADIGITS(v,p) ≥ .5 * MINVAL(p)
          then  (ABS(v) - EXTRADIGITS(v,p) + MINVAL(p)) *
                PSIGN(v,p)
       else (ABS(v) - EXTRADIGITS(v,p)) * PSIGN(v,p).
```

The following is a description of the intermediate-level functions:

EXTRADIGITS(v,p) = the absolute value described by the least significant digits left over after "fitting" into the PICTURE specification described by p

MAXVAL(p) = the maximum absolute value permitted by p

MINVAL(p) = the minimum nonzero absolute value permitted by p

PSIGN(v,p) = the sign of v when it "fits" into p.

Their formal definitions are as follows:

EXTRADIGITS(v,p) =

$$\mathrm{MOD}(\mathrm{ABS}(v)*10^{\mathrm{DECIMALDIGITS}(p)},\ 10^{\mathrm{TOTALDIGITS}(p)})/10^{\mathrm{DECIMALDIGITS}(p)}$$

$\mathrm{MAXVAL}(p) = (10^{\mathrm{TOTALDIGITS}(p)}-1)/10^{\mathrm{DECIMALDIGITS}(p)}$

$\mathrm{MINVAL}(p) = 10^{-\mathrm{DECIMALDIGITS}(p)}$

PSIGN(v,p) = <u>if</u> "S" ϵ p $\wedge$ v < 0 <u>then</u> -1
<u>else</u> 1 .

A single assertion language construct defined above, MAXVAL(p), can determine whether or not a SIZE ERROR has occurred: if the absolute value of an arithmetic operation exceeds MAXVAL(p), where p is the PICTURE specification of the destination.

Based on the above formal definitions, we can develop "rules of inference" (or lemmas) that allow proofs of assertions containing such functions.

Here is an example of how such functions are used in assertions, and how rules of inference can be used in simplification and proof. Let the PICTURE specifications of the variables A and B be 999 and 99, respectively. Then the verification condition for the statement "MOVE B TO A" could be described as follows:

$$P\{\text{MOVE B TO A}\}Q \equiv$$
$$P \supset Q^{A}_{\text{TRUNCATE}(B,999)} .$$

Since it is known that $0 \le B \le 99$, because of its PICTURE specification, it therefore "fits" into A without modification. This suggests a rule of inference: If the PICTURE specification of the destination in a MOVE operation subsumes the PICTURE specification of the source, then the TRUNCATE function need not be used in the verification condition. The verification condition then simplifies to

$$P\{\text{MOVE B TO A}\}Q \equiv$$
$$P \supset Q^{A}_{B} \quad .$$

As a second example, let us suppose that A and B have PICTURE specifications as above, and that C has a PICTURE specification of 99. Then the verification condition for the statement "COMPUTE A = B + C" would be as follows:

$$P\{\text{COMPUTE A = B + C}\}Q \equiv$$
$$P \supset Q^{A}_{\text{TRUNCATE}(B+C,999)} \quad .$$

Since the maximum value for the sum of B and C is 198, and minimum value is 0, then using the last rule (generalized to arithmetic expressions) we get

$$P\{\text{COMPUTE A = B + C}\}Q \equiv$$
$$P \supset Q^{A}_{B+C} \quad .$$

We present several examples of assertions that can be made about COBOL programs. To say that an array A is sorted in ascending order, we write

$$\forall X(1 \le X \le \text{LENGTH}(A)-1 \supset A(X) \le A(X+1)) \quad ,$$

or in LISP form

```
(FORALL X
    (IMPLIES
         (AND
              (LTQ 1 X)
              (LTQ X (SUBTRACT (LENGTH A) 1)))
         (LTQ
              (SELECT A (X))
              (SELECT A ((PLUS X 1))))))   .
```

If we wish to state that a particular value C occurs in array A, we write

$$\exists X(1 \le X \le \text{LENGTH}(A) \wedge A(X) = C) \quad ,$$

or in LISP form

```
(EXISTS X
     (AND
        (AND
             (LTQ 1 X)
             (LTQ X (LENGTH A)))
        (EQ
             (SELECT A (X))
             C)))   .
```

A typical input assertion to a COBOL program would give the values of the input files, and a typical output assertion would describe the relation of values in the input files to values in the output files. In this work, where input files and output files are disjoint, this is easy to do. However, in cases where a file may be open for input and output, we need a mechanism to distinguish between the initial values and current values in the file. One solution is to concatenate a special character to the file name to denote initial file values.

D. Abstract Assertions for COBOL Programs

As will be seen in the example, assertions and verification conditions for COBOL programs may be very long (the verification conditions are much longer than the program itself). A desirable goal of future verification research is to shorten the verification conditions and assertions, to enhance both understandability and provability. We have begun to explore some of these issues, and describe these explorations here. One way of doing this is to develop primitives for writing abstract assertions for COBOL programs. Abstract assertions could make the assertions more readable, since they would be shorter, but would they increase provability? There is still some doubt on that issue. We present some abstract assertion structures, together with their definitions and rules of inference.

In conventional program proving (including the approach taken in this work), the assertions deal with the values of variables, to the exclusion of their other attributes. This enables free substitution of values, but does not permit more powerful inferences to be made, as could be done when the other information is made available.

The main area of examination to date has been the property of equality. In conventional equality, the values of data items are considered, so that substitution may take place. We propose first a kind of equality between data items called "structural equality." Two data items are structurally equal if and only if either:

(1) Both items are elementary data items having the same PICTURE specification and the same number of occurrences, or

(2) Both items are group data items

  (a) That have the same number of immediate descendants, and
  (b) Whose corresponding descendants are structurally equal.

This is a static property of COBOL data items, but the programmer may wish to assert such a property in the program test. This definition will be used in later assertion structures.

We next define the notion "strong equality." Two data items are strongly equal if and only if either:

(1) Both items are elementary data items that are structurally equal and whose values are equal, or

(2) Both items are group data items that are structurally equal and whose corresponding descendants are strongly equal.

Both strong and structural equality are powerful properties to be asserted about tree-structured records.

There is a notion of equality connected to the MOVE statement, called "weak equality," a noncommutative relation among data items. Let A and B be data items, and let PICB be the PICTURE specification of B. A is said to be weakly equal to B if and only if either:

(1) Both items are elementary data items and B = TRUNCATE(A,PICB), or

(2) Both items are group data items

 (a) That have the same number of immediate descendants, and

 (b) Such that if AI is an immediate descendant of A and BI is the corresponding immediate descendant of B, then AI must be weakly equal to BI.

A similar kind of equality related to the MOVE CORRESPONDING statement may also be defined, called "corresponding equality" (this notion is not defined here).

A verification condition generator would have to know how to process assertions containing these abstract constructs, because some COBOL operations preserve these relations and other operations invalidate the relations.

The above framework can be extended to cover arbitrary relations on data items. Although this mechanism is a good way of relating properties of records, a mechanism for relating records within a file and the records of two different files would also be useful.

These mechanisms are useful simply because many COBOL programs entail the movement of data without extensive arithmetic operations on it. The abstract assertions described above capture some of the notions involved in data movement. One additional comment is that programmers who structure their programs so that the above-mentioned properties hold will probably be increasing the reliability of their programs. The effects of abstract assertions for COBOL on proof will be illustrated in future work.

E. Conclusions

We have shown that the assertion language for COBOL can be extremely simple. However, the reliance on a simple assertion language may make the assertions difficult to read, and the proofs unduly complex. Thus, an assertion language should be extensible so as to permit the abstract program properties to be stated concisely. The exact nature of the extensions to be chosen is a matter for future research.

VI SEMANTICS OF THE COBOL SUBSET

A. Introduction

Semantics of a programming language can never be completely separated from its syntax. Thus, in Section IV, the transduction grammar for the CSV has some relation to the semantics of the language (e.g., the fact that MOVE, COMPUTE, and ADD are all related, influenced the decision to transduce them to the common primitive SET$). However, most of the semantic issues are left uninterpreted since we have not yet defined the semantics of Transduced COBOL. We define these semantics by describing each of the primitives in Transduced COBOL in terms of a simple language involving only assignments, tests, and branches. The semantics of this simple language are in turn described by the way in which verification conditions are generated for programs in it. Thus, the semantics of the COBOL subset are contained in the description of the operation of the Posttransduction Processor and the Verification Condition Generator. Both programs have been abstractly described in Section III of this report. In this section, we present a more detailed description of both programs, followed by a discussion of some research issues in verification condition generation.

B. Posttransduction Processing

Posttransduction processing:

(1) Transforms the label structure of the program so that paragraph names are unique. In COBOL, two different sections may have paragraphs of the same name. The section structure may then be taken out.

(2) Forms a list of labels with their corresponding assertions for later use by the path analyzer (part of the verification condition generator) for processing GO TO statements.

(3) Eliminates the paragraph structure of the program. A copy of the program, after step (1), has been saved so that labels can be processed correctly.

(4) Scans each sentence in the program and translates it into the simple language to generate a verification condition. The sentences transformed are:

(a) PERFORM statements

(b) All I/O statements

(c) All assignment statements

(d) All CORRESPONDING statements.

These operations need knowledge of the symbol table as well as knowledge of the transduced program. We now present some simple examples of the kind of operations performed in posttransduction processing.

Suppose that we start with a very simple COBOL program as follows:

```
PROCEDURE DIVISION.
S1   SECTION.
P1.
     (ASSERT (GTQ X 5)).
     ADD 1 TO X.
P2.
     ADD 2 TO X.

S2   SECTION.
P3.
     ADD 1 TO X.
P1.
     ADD 1 TO X.
     (ASSERT (GTQ X 10)).                        (VI-1)
```

Its transduced form would be:

```
[PROCEDUREDIVISIONS$ (SECTION$ S1 (PARAGRAPH$ P1
                                              (ASSERT (GTQ X 5))
                                              (SET$ X
                                                    (PLUS X 1)
                                                    NIL))
                                  (PARAGRAPH$ P2 (SET$ X
                                                    (PLUS X 2)
                                                    NIL)))
                     (SECTION$ S2 (PARAGRAPH$ P3
                                              (SET$ X
                                                    (PLUS X 1)
                                                    NIL))
                                  (PARAGRAPH$ P1 (SET$ X
                                                    (PLUS X 1)
                                                    NIL)
                                           (ASSERT (GTQ X 10]
```

The first stage of posttransduction processing creates unique labels as follows:

```
[PROCEDUREDIVISION$ (SECTION$ S1 (PARAGRAPH$ (P1 S1)
                                             (ASSERT (GTQ X 5))
                                             (SET$ X
                                                   (PLUS X 1)
                                                   NIL))
                                 (PARAGRAPH$ (P2 S1)
                                             (SET$ X (PLUS X 2)
                                                   NIL)))
                    (SECTION$ S2 (PARAGRAPH$ (P3 S2)
                                             (SET$ X
                                                   (PLUS X 1)
                                                   NIL))
                                 (PARAGRAPH$ (P1 S2)
                                             (SET$ X (PLUS X 1)
                                                   NIL)
                                             (ASSERT (GTQ X 10]
```

Notice how the two paragraphs named P1 may now be distinguished because their section names have been joined with them into a list. The sections are then taken out:

```
[(PARAGRAPH$ (P1 S1)
             (ASSERT (GTQ X 5))
             (SET$ X (PLUS X 1)
                   NIL))
 (PARAGRAPH$ (P2 S1)
             (SET$ X (PLUS X 2)
                   NIL))
 (PARAGRAPH$ (P3 S2)
             (SET$ X (PLUS X 1)
                   NIL))
 (PARAGRAPH$ (P1 S2)
             (SET$ X (PLUS X 1)
                   NIL)
             (ASSERT (GTQ X 10]
```

In the second stage, all paragraph names associated with assertions are listed to be used later in processing GO TO statements. The list is not needed for this program but is made anyway. The list is:

```
(((P1 S1) ASSERT (GTQ XX 10)))  .
```

The paragraph structure of the program can now be eliminated as follows:

```
((ASSERT (GTQ X 5))
 (SET$ X (PLUS X 1)
       NIL)
 (SET$ X (PLUS X 2)
       NIL)
 (SET$ X (PLUS X 1)
       NIL)
 (SET$ X (PLUS X 1)
       NIL)
 (ASSERT (GTQ X 10)))
```

Now each sentence in the program is translated into its equivalent form in a simpler language called "Posttransduced COBOL." The example program translates to:

```
((ASSERT (GTQ X 5))
 (ASSIGN X (TRUNCATE 999 (PLUS X 1)))
 (ASSIGN X (TRUNCATE 999 (PLUS X 2)))
 (ASSIGN X (TRUNCATE 999 (PLUS X 1)))
 (ASSIGN X (TRUNCATE 999 (PLUS X 1)))
 (ASSERT (GTQ X 10)))
```

In this case the assignment statements were augmented to include truncation (note that the PICTURE specification of X is 999).

We now present examples of how other statements in the language are translated into posttransduced form. The most complicated is the PERFORM statement. We have decided to handle the paragraphs that make up the body of the PERFORM statement by expanding them in-line. Another way to handle the body of a PERFORM statement is to treat it as a procedure call, with entry and exit assertions describing the effects of the PERFORM statement. There is a clear trade-off here: in simple programs (without many PERFORM blocks that are repeatedly used) the expansion method is preferable, because there are fewer proofs to generate; in more complex programs the procedure-call method is preferable, because the proof of the PERFORM body need only be done once even if the PERFORM block is used many times. The most interesting options are the PERFORM VARYING and PERFORM n TIMES because they are loops that must be translated into assignments, tests, and branches. For example, the COBOL statement

```
PERFORM P1 VARYING I FROM 1 BY 1
    UNTIL I > N (ASSERT (P I))  .
```

would be translated, as shown in Figure VI-1. There is an initialization

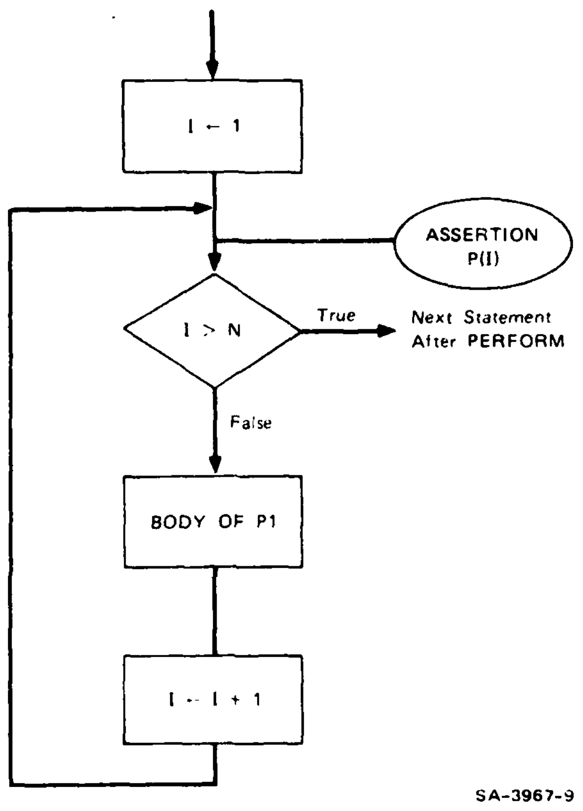


FIGURE IV-1 TRANSLATION OF "PERFORM P1 VARYING I FROM 1 BY 1 UNTIL I > N (ASSERT (P I))"

of I, a list on I and the increment of I. The COBOL statement

```
     PERFORM P1 N TIMES.
```

would also be translated in the same way. As a detailed example, let us examine the following COBOL program

```
     PROCEDURE DIVISION.

     P1.
          MOVE 0 TO SUM.
          PERFORM P2 VARYING I FROM 1 BY 1
               UNTIL I > N (ASSERT (GTQ SUM 0)).
          STOP RUN (ASSERT (GTQ SUM 0)).

     P2.
          ADD A(I) TO SUM.
                                                      (VI-2)
```

Its transduced form is as follows:

```
  [PROCEDUREDIVISION$ (SECTION$ FIRSTSECTION
                                [PARAGRAPH$ P1 (SET$ SUM 0 NIL)
                                            (PERFORM
                                              VARYING
                                              (DO$ P2 P2)
                                              (I 1 1 (GT I N))
                                              (ASSERT (GTQ SUM 0)))
                                            (STOP (ASSERT (GTQ SUM 0]
                                (PARAGRAPH$
                                  P2
                                  (SET$ SUM (PLUS (SELECT A (I))
                                                  SUM)
                                        NIL]
```

After posttransduction processing it looks like this:

```
[(ASSIGN SUM (TRUNCATE 999 0))
 [BLOCK (ASSIGN I (TRUNCATE 99 1))
        (ASSERT (GTQ SUM 0))
        (IF (GT I N)
            (ENDPERFORM)
            (NEXT))
        (ASSIGN SUM (TRUNCATE 999 (PLUS (SELECT A (I))
                                        SUM)))
        (ASSIGN I (TRUNCATE 99 (PLUS I 1)))
        (LOOPASSERT (ASSERT (GTQ SUM 0]
  (STOP (ASSERT (GTQ SUM 0)))
  (ASSIGN SUM (TRUNCATE 999 (PLUS (SELECT A (I))
                                  SUM]
```

Notice how the body of P2 has been expanded and the initialization, increment, and test have been included. The loop assertion appears in two places: at the beginning of the loop, and as part of the loop (in the LOOPASSERT statement). The statement ENDPERFORM indicates that control is to be passed to the statement following the PERFORM statement. If P2 had more than one statement, then the expansion would be in terms of a list headed by the keyword BLOCK, indicating multiple statements.

All input-output statements must be translated to array accesses. A sequential file F is represented as a set of arrays (__.ARRAY)--one for each elementary item in the record description. There is an array pointer (F.INDEX) that indicates the record currently being processed. A variable F.LENGTH indicates the number of records in the file. READ and WRITE simply perform MOVE operations from the array to and from the file's record in the program and increment the array pointer. OPEN and CLOSE simply set the array pointer. For example, the COBOL statement

```
                    OPEN INPUT X.
```

has as transduced form

```
                    (OPENINPUT$ X).
```

In posttransduction processing it becomes

```
                    (SET$ X.INDEX  0).
```

Note that X.INDEX is the array pointer. The COBOL statement

```
                    READ  X  AT END  GO TO P1.
```

transduces to

```
                    (READ  X  (GO  P1)).
```

Suppose that Y is the record for file X, and the data declaration for Y is as follows:

```
    01      Y.
            02  Z1  PICTURE 999.
            02  Z2  PICTURE S9V999.
```

Then there are to be two arrays: Z1.ARRAY and Z2.ARRAY. The number of records in the file is represented by the variable X.LENGTH. The intermediate form of the statement before translation of the assignment statements is

```
(SET$ X.INDEX (PLUS X.INDEX 1))
(IF (GT X.INDEX X.LENGTH)
    (GO P1)
    (NEXT))
(SET$ Z1 (SELECT Z1.ARRAY(X.INDEX))
(SET$ Z2 (SELECT Z2.ARRAY(X.INDEX)).
```

Note that the AT END option is a test to see whether the current index is greater than the number of records.

Assignment statements are transformed by using the function TRUNCATE(p,e)--p is a PICTURE specification and e is an expression--to truncate the assigned expression to the PICTURE specification of the destination data item. If the assignment statement has the ROUNDED OPTION, then the function ROUND(p,e) replaces TRUNCATE. The SIZE ERROR option is transformed into an IF statement, testing the absolute value of the expression against MAXSIZE(p), where p is the PICTURE specification of the destination. For example, the COBOL statement

```
COMPUTE X = Y + Z ON SIZE ERROR GO TO P1.
```

transduces to

```
(SET$ X
      (PLUS Y Z)
      (GO P1)).
```

Suppose the PICTURE specification of X is S999V9. Then the posttransduced form looks like this:

```
(IF (GT (ABS (PLUS Y Z))
        999.9)
      (GO P1)
      (NEXT))
(ASSIGN X (TRUNCATE S999V9
                     (PLUS Y Z))).
```

If no SIZE ERROR clause is specified, there is no IF statement. This is slightly at variance with COBOL 74, since it specifies that if a SIZE ERROR condition occurs and no SIZE ERROR clause is specified, then no assignment occurs. We intend to remedy this inconsistency in future work.

The CORRESPONDING operations have a particularly interesting transformation to posttransduced form. The definition of MOVE CORRESPONDING A TO B is as follows:

(1) If A is an elementary data item, MOVE A TO B.

(2) Otherwise take all immediate descendants of A that have the same name as any immediate descendants of B, and put them in set S. For all elements X in S, MOVE CORRESPONDING X OF A TO X OF B.

As an example, suppose A and B have the following data declarations:

```
01  A.
     02 C.
        03 E PICTURE 999.
     02 D.
        03 G PICTURE 999.
     02 F PICTURE 999.
01  B.
     02 C.
        03 G PICTURE 999.
        03 E PICTURE 999.
     02 F PICTURE 999.
```

The statement

```
MOVE CORRESPONDING A TO B.
```

transduces to

```
(MOVECORRESPONDING$ A B).
```

After posttransduction processing this becomes the two statements

```
(ASSIGN (QUAL E C B)
        (TRUNCATE 999 (QUAL E C A)))
(ASSIGN (QUAL F B)
        (TRUNCATE 999 (QUAL F B))).
```

C. Verification Condition Generation

The verification condition generator is given the posttransduced COBOL program as input. Verification condition generation has two stages:

(1) Analysis of all the simple paths through the program. A simple path is a program path that has an entry

assertion, an exit assertion and a fixed number of program statements in between. A list of these simple paths is returned by the path analyzer. The path analyzer must have semantic knowledge of the following posttransduced statements:

(a) IF
(b) GO TO
(c) BLOCK (multiple statements in posttransduced program)
(d) ENDPERFORM
(e) LOOPASSERT
(f) STOP

The statements in a simple path are presented backwards relative to the order of execution. This is necessary for the next stage.

(2) Creation of the verification condition from the path description. The exit assertion is pushed backwards through the program path. This involves substition when an assignment statement is encountered, and the construction of implications when assertions or IF statements are encountered. The final verification condition is returned at this stage.

In path analysis all IF statements generate two possible paths--one for instances when the condition is true and another for instances when the condition is false. The condition that holds for a paticular path (either true or false) becomes part of the path as an argument to the IF statement. Thus, the COBOL statements,

```
COMPUTE X = Y + Z.
IF X > 0 NEXT SENTENCE
     ELSE ADD 1 to X.,
```

transduced and posttransduced as follows:

```
(ASSIGN X (TRUNCATE 999
                    (PLUS Y Z)))
(IF (GT X 0)
    (NEXT)
    (ASSIGN X (TRUNCATE 999
                        (PLUS X 1)))),
```

would generate the two partial paths

```
((IF (GT X 0))
 (ASSIGN X (TRUNCATE 999
          (PLUS Y Z))))
```

and

```
((ASSIGN X (TRUNCATE 999
                    (PLUS X 1)))
 (IF (NOT (GT X 0)))
 (ASSIGN X (TRUNCATE 999
                    (PLUS Y Z)))
```

for the true and false conditions of the IF, respectively. Note the reverse order of the statements.

A GO TO statement forms the end of a path, and the assertion attached to the label of the destination paragraph must be fetched. That assertion would be found on the global variable LABELASSERTLIST described in the previous subsection. For example, if there is a posttransduced statement like this (paragraph L1 occurs only in section S1)

```
      .
      .
      .
   (GO  L1)
```

and LABELASSERTLIST has an entry

```
((L1 S1) ASSERT (LT P Q)),
```

then that assertion would be included at the end of any path that ended with a branch to L1.

All statements appearing in a BLOCK are simply processed individually. An ENDPERFORM statement generates the exit path from the PERFORM. A LOOPASSERT statement generates the path around a PERFORM loop. STOP simply ends that path.

One example of output from the path analyzer is from program VI-1:

```
[((ASSERT (GTQ X 10))
  (ASSIGN X (TRUNCATE 999 (PLUS X 1)))
  (ASSIGN X (TRUNCATE 999 (PLUS X 1)))
  (ASSIGN X (TRUNCATE 999 (PLUS X 2)))
  (ASSIGN X (TRUNCATE 999 (PLUS X 1)))
  (ASSERT (GTQ X 5]
```

Since it is a straight-line program, there is only one path. Another example is from program VI-2:

```
[((ASSERT (GTQ SUM 0))
  (IF (GT I N))
  (ASSERT (GTQ SUM 0)))
 ((ASSERT (GTQ SUM 0))
  (ASSIGN I (TRUNCATE 99 (PLUS I 1)))
  (ASSIGN SUM (TRUNCATE 999 (PLUS (SELECT A (I))
                                  SUM)))
  (IF (NOT (GT I N)))
  (ASSERT (GTQ SUM 0)))
 ((ASSERT (GTQ SUM 0))
  (ASSIGN I (TRUNCATE 99 1))
  (ASSIGN SUM (TRUNCATE 999 0]
```

This program is a single-loop program and therefore has three paths. The first path is the exit path from the program when the PERFORM is finished. The second path is the loop path. The third path is the initialization path. In both programs, the paths are listed in reverse order of execution.

In verification condition generation a path is converted into a formula to be proved. The formulae for all the paths are conjoined together, yielding the verification condition for the entire program. The verification condition generator moves through the path (listed backwards by the path analyzer) building the formulae as it goes. Let x be an arbitrary Boolean expression. If it encounters an (ASSERT x) or (IF x) and the formula is f, then the new formula is (IMPLIES x f). If it encounters an (ASSIGN V e), then the new formula has e substituted everywhere for V.

One example of a completed verification condition comes from program VI-1:

```
(AND
  (IMPLIES
    (GTQ X 5)
    (GTQ
      (TRUNCATE
        999
        (PLUS (TRUNCATE 999
                        (PLUS (TRUNCATE 999
                                        (PLUS (TRUNCATE 999
                                                        (PLUS X 1))
                                              2))
                              1))
              1))
      10)))
```

This formula is valid, because if X is no less than 5, then adding 5 to X will make X no less than 10. The only difficulty is if truntion takes place, but X will always be truncated to 999. Q.E.D.

The second example comes from program VI-2:

```
(AND (IMPLIES (GTQ SUM 0)
              (IMPLIES (GT I N)
                       (GTQ SUM 0)))
     (IMPLIES (GTQ SUM 0)
              (IMPLIES (NOT (GT I N))
                       (GTQ (TRUNCATE 999 (PLUS (SELECT A (I))
                                                SUM))
                            0)))
     (GTQ (TRUNCATE 999 0)
          0))
```

There are three conjuncts (conditions) to be proved. The first condition (the exit condition) is trivially true: If SUM is no less than 0, then if I is greater than N, then SUM is no less than 0. The second condition cannot be met, because even if SUM is no less than 0, adding A(I) to it could make it less than 0. The third condition is trivial: 0 is no less than 0. However, since the second condition could not be met, the program could not be proved correct. Imposing stronger conditions at the loop and initialization points, such as $\forall I(A(I) \geq 0)$, would enable the proof of this simple program.

Note that a program can fail to be verified for three reasons:

(1) The program is wrong--i.e., it has a bug.

(2) The input/output assertions (the specifications of the program) are wrong.

(3) The program and input/output assertions are mutually consistent, but the intermediate assertions have been chosen incorrectly.

Only the programmer (not the deductive system) can determine which of these is the reason for a program's failure to be proved. However, a good deductive system may be able to generate a counterexample to enable the programmer to identify the trouble.

D. Research Issues in Verification Condition Generation (or Posttransduction Processing)

All possible improvements to verification condition generation (and posttransduction processing) would be of one kind: add knowledge to the system to make the verification conditions simpler. In other words, perform some of the proof effort early. The reason for this is that it may be easier to perform some simplification when the knowledge is more readily applicable. For example, elimination of the TRUNCATE operation could be easily performed when the TRUNCATE operator is inserted (by means of a simple test); but when a deductive system is in operation, such a simplification could not be easily made because knowledge of the maximum size of the source item would be lost.

In addition to ordinary simplification, verification condition generators might be designed to handle abstract assertions about COBOL programs (described in Section V). To do this, a verification condition generator would have to know which abstract assertions are preserved by which program statements.

We have yet to determine the exact nature of the gain to be made by doing simplification during the verification condition generation process. However, we are hopeful that it will yield major improvements in the efficiency of the program verification process.

VII STRUCTURE AND COBOL VERIFICATION

A. Introduction

We consider four aspects of structure related to the verification of COBOL programs:

(1) Use of structured control primitives

(2) Restrictions on COBOL operations

(3) Use of data bases in constructing large COBOL systems

(4) Generalized facilities for data abstraction

(5) Top-down design and modularity.

All of these structuring facilities have the goal of reducing the complexity of the program by breaking it up into manageable units.

B. Structured Control Primitives

The use of a limited set of "well-structured" control primitives in writing programs is the practice commonly known as "structured programming." Instead of the normal COBOL control constructs, the programmer writes in a block-structured, lexically nested medium (see Section II-G on control statements) using only the following constructs (as an example):

(1) IF b THEN s1 ELSE s2

(2) WHILE b DO s1

(3) DO s1 UNTIL b

(4) FOR v = e1 TO e2 BY e3 DO s1

(5) CASE e1 OF s1,...,sn.

The semantics of the above verbs have been described in the literature on structured programming. In the above expressions, b is an arbitrary condition, si (for all i) is either a single statement or a sequence of statements preceded by BEGIN and followed by END, v is an arbitrary variable, and ei (for all i) is an arbitrary expression. The resulting programs are much easier to read and, on the average, simpler than programs written using the standard COBOL control primitives. By simplicity we mean having a small number of control paths through the program. However, for any program written using structured programming primitives, an equivalent program that is equally simple can be written using the standard COBOL control primitives. The reason for this is that the complexity of program verification depends on three variables:

(1) The number of simple paths in the program
(2) The complexity of the assertions
(3) The number of statements per simple path.

No change in any of the above variables is made simply by changing from a more restricted set of control primitives (the structured ones) to a less restricted one (those of COBOL 74). However, we endorse the use of structured programming primitives, because they tend to lead to simpler (and thus easier to verify) programs. We also endorse the practice of training programmers to write programs with nested control schemes (flowcharts) using the primitives of COBOL 74. This is a necessary first step towards the improvement of software reliability in a COBOL job shcp.

C. Restrictions on COBOL Data Operations

In the analysis of COBOL (Section II) and the presentation of the subset (Section IV), we mentioned that certain data operations were not amenable to verification. They were:

(1) Automatic truncation of the most significant digits on a NUMERIC MOVE operation.
(2) Allowing the compiler to permit data operations from one type to another that could later yield a type error, such as moving an ALPHANUMERIC value to a NUMERIC data item.

In the first case, the language permits the programmer to make a mistake (e.g., accidentally truncate the most significant digits) because it is syntactically the same as a correct operation. A solution to this problem is to syntactically differentiate a MOVE operation in which significant digit truncation is intended, for example

```
MOVE TRUNCATED A TO B.
```

In the second case, the language permits an operation for which dynamic type-checking should take place, but does not. Dynamic type-checking, although inefficient, should be imposed, and a special statement should be added to the language, such as

```
MOVE NUMERIC A TO B; ON TYPE ERROR PERFORM P1.
```

(if A is non-NUMERIC and B is NUMERIC). Just as READ operations have the AT END clause and arithmetic operations have the ON SIZE ERROR clause, so should the special MOVE statements have an ON TYPE ERROR clause.

The suggested restrictions would make COBOL programs more reliable, and would make verification easier (because the verifier must check for type errors even if the run-time system does not).

D. Data Bases

The files that are modified by a COBOL program or a system of COBOL programs can be considered as a single large data base. Thus, the use of a data base management facility in conjunction with a system of COBOL programs, has received much attention recently. In fact, the CODASYL Data Base Task Group[14] has produced a structure in which the data description language is almost exactly like the COBOL 74 DATA DIVISION, and the application programs for the data base are almost exactly like the COBOL PROCEDURE DIVISION.

What do we gain by using this approach? If one looks at the file layout for a typical COBOL system (see Figure VII-1), there are many programs, many files, and some files that are shared by more than one program. The design decisions involved in the layout of the data are scattered throughout the ENVIRONMENT and DATA DIVISIONS of all the programs. The data base management system (DBMS) approach centralizes all the decisions on the data layout in the scheme for the DBMS. The declarations for the data visible to each program are the subschemas and are also part of the DBMS. The subschemas must be consistent with the schema. However, this approach allows the distinction between physical data declaration (in the schema) and logical data declaration (in the subschemas). With this approach decisions can be hidden from the programs that do not need to know about it. A DBMS is a good tool for the development of large COBOL systems because it separates the systems analysis (layout of data and programs) phase of the system development from the coding phase, something that has always been done in practice but has never been enforced by the available automated tools. We believe that the DBMS approach will increase the reliability of the systems produced by using it (with a possible reduction in efficiency depending on the DBMS system characteristics).

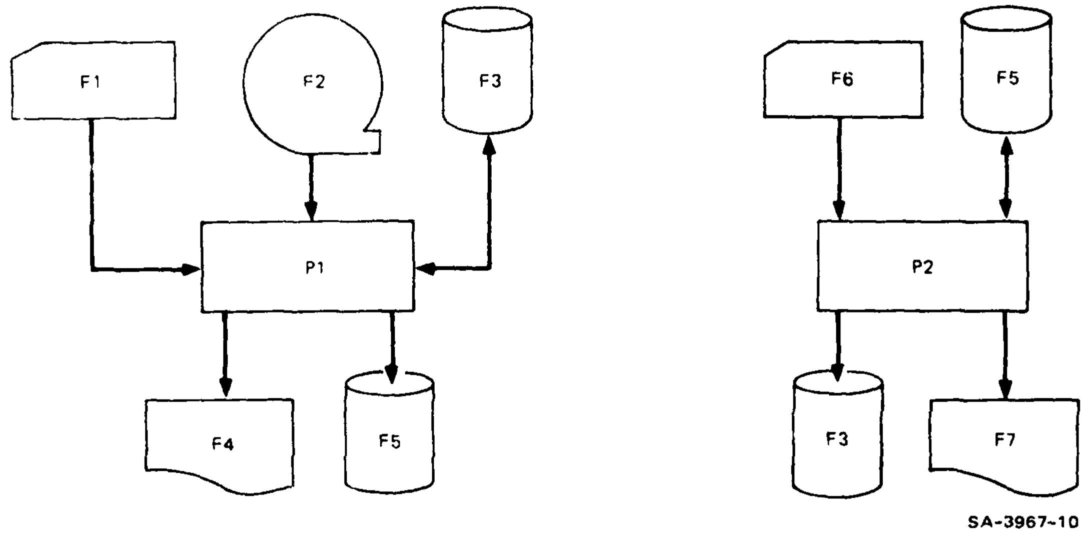


FIGURE VII-1 FILE STRUCTURE FOR A TYPICAL COBOL SYSTEM

We anticipate no difficulty in using the techniques developed here for verification of programs using the CODASYL DBMS. However, we also do not anticipate an inherent gain in the ease of verification simply from using a DBMS approach, other than that systems designed in this method will be better structured and therefore simpler. This is because a DBMS approach does not reduce the complexity of either the programs or the assertions in a COBOL system. One intuitive reason for not gaining is that decisions on data declarations and representations are still shared by the PROCEDURE DIVISIONs of all the programs in the COBOL system.

E. Data Abstraction Facilities

In the last two years, much attention has been given to the issue of data abstraction. In its simplest form data abstraction associates a set of abstract operations with a set of abstract data objects (called a type). All programs wishing to manipulate objects of an abstract type must call the operations. The calling programs do not need to know the representation for an abstract object, only its formal properties (described in terms of assertions written at a level of abstraction appropriate to the abstract objects being manipulated). Thus, for example, the users of a stack need only know about the formal properties of PUSH and POP (tne stack's operations) and not about the implementation of a stack in terms of an array (to store the stack) and a variable (to store the stack pointer). Some researchers[19,20] advocate special languages to facilitate the use of data abstraction, while others[1] emphasize a formal medium for describing the properties of a data abstraction. However, most researchers agree that the method promises to reduce the complexity of program verification for the following reasons:

(1) The programs are less complex. A single-level program can be broken up into two levels of simpler programs--one level that manipulates abstract objects and another level that implements them. Since complexity of programs is considered to vary exponentially with their length, it is probably easier to prove many simple programs than a few complex ones. There may in fact be less code to prove correct, because the code for a data abstraction may have been previously duplicated, but the abstraction process has put it all in one place. Successive decompositions into two levels can lead to systems with many levels.

(2) The assertions are simpler. This is probably the most significant property of data abstraction. Since the abstract assertions typically contain less information (implementation variables and algorithms are hidden), they are shorter and thus are easier to manipulate. On the other hand, the implementation assertions do not have to deal with how the abstract resource is being used and can typically be simpler. For example, keyed records can be implemented either by a hash table or a linked list. The abstraction is simply a function from keys to records--all the information about links and hashing functions is eliminated from the higher level.

Data abstraction has shown some benefits in practical system design. In another SRI project [2,21] a general-purpose operating system (whose major design goal is verifiable security) was designed using a formal methodology based on data abstraction. The operating system has 13 levels of data abstraction. A major question is whether the kinds of systems typically designed using COBOL are amenable to data abstraction. Operating systems have abstractions such as virtual memory segments, file directories, and processes; but it seems that no such abstractions are visible within COBOL systems. More research must be done on this matter to determine whether COBOL programs can take advantage of this valuable technique. If so, we suggest a preprocessor that allows the distinction between programs that use a data abstraction and programs that implement one. The two kinds of programs can be combined in some manner (perhaps by macro expansion) before the system is run.

Many sophisticated languages (including COBOL) have some data abstraction built into their basic facilities. Things such as indexed sequential files and the hiding of input/output buffering are examples of data abstraction. Some of COBOL's facilities, such as the elementary data item, permit abstraction, but are circumvented by other facilities (such as REDEFINES). In a good data abstraction, the implementation details are completely hidden from the user of the abstraction.

F. Top-down Design and Modularity

Top-down design is an embodiment of the philosophy that says, "When there are many decisions to make in the construction of a program, make the most important ones first." This is useful because each new decision that is made must be consistent with the ones made before. Thus, the earlier decisions tend to influence the later ones, and not the reverse. It would be unfortunate if the lesser decision would influence a more important one. Data abstraction is one embodiment of this philosophy because the decision concerning the formal properties of a data abstraction is made before the decision of how to implement it.

Top-down design, as advocated by Dijkstra[22] and Mills[23] begins with a statement of the problem to be solved, together with a very abstract program to solve the problem. The program typically has control statements interspersed with natural language. For example, to sort an array of length N:

```
for i ← 1 to N-1 do
    swap the smallest element in A[i]...A[N]
         with A[i];
```

The programmer then makes successive refinements in the abstract program, incorporating new decisions as they are made, until the final program is achieved. These decisions can be made concerning either data or algorithms. This approach seems useful for COBOL programs in deciding the algorithms, but not the data, because a COBOL program typically consists of many programs operating on data that has probably been decided on beforehand (unless some of the most abstract programs were done during the systems analysis phase). In general, this is a good way to write programs, because it increases a programmer's understanding of the programming problem. As Dijkstra indicates, a programmer doing stepwise refinement can make "convincing" arguments for the correctness of a refinement. If each refinement is correct and consistent with the preceding ones, then the correctness of the entire program can be inferred. However, it is impossible to formally prove the correctness of any program written in a nonformal medium, such as natural language interspersed with control statements. Thus, top-down design shows little promise in making COBOL verification easier, except that it may lead to better programs.

Modularity is an embodiment of two beliefs

(1) Put all related design decisions in one place

(2) Put all separable design decisions in different places.

The result is a set of clusters of decisions. Each cluster of decisions is made only in a distinct module of the system. Data abstraction is a good example of modularity because the related decisions (concerning the maintenance of a data abstraction) are all together in a related set of programs, which is in turn separated from all the programs that use the data abstraction. Just because a system is broken up into small programs does not mean that it is modular. For example, a program A that operates on five files may be modularized into two smaller programs--one operating on two files and the other operating on three files. This decomposition is probably modular. However, if A is broken up into two smaller programs each operating on five files, it may or may not be a modularization. An attempt to modularize a large COBOL program will yield benefits only if it shortens or simplifies the assertions of the larger program. One way to assist this is to describe, for each module, the set of data items accessed and modified by the module. This may enable the verification system to simplify the assertions used in the correctness proof of the module.

G. Conclusions

Some structuring disciplines seem especially promising in decreasing the cost of program verification, and COBOL verification in particular. Specific progress in implementing any of the schemes described above has yet to be made, and is the subject for future research in this area.

VIII AN EXAMPLE OF COBOL VERIFICATION

A. Description of Program

In this section we present a COBOL program that we have taken through the several stages of machine processing to generate verification conditions. One of the verification conditions is proved in its entirety, and the proofs of the others are sketched.

We have verified a payroll program, which is simple relative to the complexity of an average COBOL program. The complexity of the program, which is at the upper bound of practical program verification, will be revealed in this section. The number of relevant cases is very large, which is reflected in the number of execution paths through the program.

The function of the program is to update a master file and print the employees' weekly paycheck, given a master file, containing the employees' cumulative payroll information, and a time card, giving the number of hours worked in a week. The file structure of the program is shown in Figure VIII-1.

The structure of the program at the highest level is as follows, with appropriate paragraphs in parentheses:

(1) File opening and initialization (OPEN-FILES)

(2) Main processing loop (LOOP)

(3) Files closing (CLEANUP).

The main processing loop consists of three parts:

(1) Reading files (READ-INPUT-MASTER through READ-TIME-CARD) and checking for end-of-file (GO TO CLEANUP) and file structure errors (GO TO ERROR-ABORT)

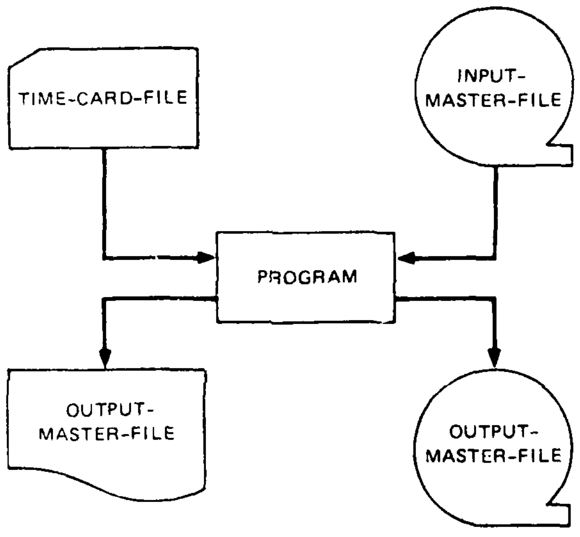
(2) Processing a related set of records (PROCESS-RECORDS)

(3) Writing the output records (WRITE-OUTPUT).

CLEANUP and ERROR-ABORT terminate the program. Record processing has the following components:

(1) Setting up the output master record (MOVE-ARRAY) with possible error conditions (GO TO ERROR-ABORT)

(2) Setting up the paycheck record.



SA-3967-11

FIGURE VIII-1 FILE STRUCTURE OF THE EXAMPLE PROGRAM

MOVE-ARRAY sets up a single element of the array in the output master record, and must be PERFORMed in a loop. ERROR-ABORT terminates the program.

The program is shown in Table VII-1*.

B. Assertions

The numbered assertions of the program occur in the following places:

(1) Input to the program

(2) The beginning of the main processing loop

(3) Normal exit from the program

(4) Error exit from the program--file structure error

(5) Error exit from the program--record error or overflow

(6) Loop for moving the array information.

The assertions are written out (by number as above) in Table VIII-2.

The input Assertion (1) states that the input files are the same length, that their name fields correspond, and that the array pointer of each record is within bounds. Assertions (2), (3), and (6) use the general invariant for the program, Assertion (7), which states that the input assertion holds and that for each corresponding file record

(1) The output value for the name equals the corresponding input value.

(2) The output gross pay has been correctly calculated from the input gross pay, the salary, and the hours worked in the current week.

(3) The output hours worked is the sum of the input hours worked and the hours worked in the current week.

(4) The current week has been incremented from input to output.

(5) For all members of the weekly hours array (except the current week), the input value equals the output value.

(6) The output element of the weekly hours array for the current week equals the hours worked this week.

*The tables are at the end of the section.

This is the "essence" of the workings of the program. Assertion (2) states that the double end-of-file condition has not occurred and that all the file pointers are equal. Assertion (3) states that the double end-of-file has occurred and that the lengths of the input files and of the output files are equal. Assertion (6) states that the pointers to the input files are one greater than the pointers to the output files, that all the variables moved before the loop have been properly processed, and that I-1 of the array elements have been processed correctly.

Assertion (4) is FALSE. Thus, we must prove that this particular error condition (having the end-of-file without the other) never happens. Assertion (5) states that either a rounding error or a name correspondence error has occurred.

C. Transduction and Posttransduction Processing

The transduced PROCEDURE DIVISION appears in Table VIII-3. The transduced DATA DIVISION appears in Table VIII-4.

Posttransduction processing was then undertaken. Several hand-simplifications were made after posttransduction processing: the redundant statements were removed (expansion of a PERFORM produces a block of code--the original--that is never executed); all unneeded TRUNCATE statements were removed (we purposely wrote the program so there would be no truncation on assignment statements); all redundant qualifications were removed. The resulting program appears in Table VIII-5.

D. Verification Condition Generation

The program of Table VIII-5 has 26 simple paths through it. The large number arises because of the number of IF statements (without transfers) in the program. For example, if there is a simple path from Assertion (2) to Assertion (6) that has three IF statements in it (without transfers), then there are eight (2^3) possible ways to take the path, depending on the disposition of each of its constituent IF statements. When an IF statement has a GO TO in it, a new path is generated. For the program under study, the number of simple paths are as follows (listed by source and destination assertion number):

1-2	1
2-3	4
2-4	4
2-5	8
2-6	4
6-6	1
6-2	1
3-3	1
4-4	1
5-5	1 .

Fortunately many of these paths can be eliminated from consideration, because they are never taken in actual execution of the program. A simple path of a program is never taken if and only if:

(1) The antecedent of the VC (verification condition for the path) is FALSE.

(2) All paths that lead to the beginning of the path in question are never taken. For example, two of the error conditions tested for by the program check whether the files are not of the same length (2-4) and whether the corresponding NAME fields are not equal (2-5). These two conditions can never occur, because Assertion (2) states that they do not happen. The remaining eight paths are as follows:

1-2	1
2-3	1
2-5	1
2-6	1
6-6	1
6-2	1
3-3	1
5-5	1 .

Of the remaining paths, two (the last two) are trivial, because the input and output assertions are identical, with no intervening code. Thus, the VC looks like $P \supset P$, a trivial deduction. The output of path analysis after the elimination of impossible and trivial paths is shown in Table VIII-6.

There is much to be learned from the hand-simplifications performed after posttransduction processing and path analysis. The amount of material would have been exceedingly long without the simplifications. From this we conclude that incremental simplification at all stages of the verification process is useful in being able to keep the volume of material to the level of understandability. We may also conclude that the level of complexity (before simplification) of even such a simple program is large indeed. This is one clue to the complexity of software--and how it is related to the complexity of program verification.

Simplification could work in this example because the program was written so as to be amenable to such optimization of the verification process:

(1) All MOVE statements were written so that truncation would not be needed.

(2) Most names in the program were chosen to avoid the necessity of qualification.

(3) The control structure of the program was well-designed so that superfluous paths would not be taken.

Just as there are techniques for writing understandable, modifiable programs, there are also techniques for writing verifiable programs.

All the verification conditions (VCs) are not written because they are verbose. However, we state what each VC means and why it is true:

- (1-2) This VC states that the input assertion and effects of file opening (initializing the file pointers to 0) is sufficient to prove the null (initial) instance of the program invariant. This is true because no records have been processed.

- (2-3) This VC states that if n records have been processed correctly, the effects generating two ends of file in reading one input record guarantee that the output assertion is true: n records have been correctly processed, where n is the number of records in both input files. The conclusion is simply a restatement of the hypothesis.

- (2-5) This VC states that if n records have been processed correctly, then the effects of generating a SIZE ERROR in processing the $(n+1)^{st}$ record, guarantee that either a NAME mismatch or a SIZE ERROR must have occurred in processing one of the records.

- (2-6) This VC states that if n records have been processed correctly, the effects of correctly processing some of the $(n+1)^{st}$ record guarantee that:

  (a) n records have been processed correctly

  (b) some of the $(n+1)^{st}$ record has been processed correctly

  (c) 0 elements of the array have been processed correctly.

. (6-6) An induction--if

(a) n records have been processed correctly

(b) some of the $(n+1)^{st}$ record has beer. processed correctly

(c) m elements of the array have been processed correctly

then the effects of processing correctly the $(m+1)^{st}$ array element guarantee (a), (b), and that (m+1) elements of the array have been processed correctly.

. (6-2) If

(a) n records have been processed correctly

(b) some of the $(n+1)^{st}$ record has been processed correctly

(c) 52 elements of the array have been processed correctly

then the effects of processing the remainder of the record correctly and writing the output files guarantee that (n+1) records have been processed correctly.

E. Proof of a Verification Condition

We now sketch the proof of the verification condition for path (6-6), which is presented in Table VIII-7. The complete proof of this verification condition is contained in Table VIII-8, and the rules of inference for the proof are contained in Table VIII-9. Formal background for the proof can be found in any textbook in mathematical logic (e.g., Reference 25). The structure of the verification condition is as follows:

```
(AND
   (IMPLIES  a
      (IMPLIES  b
                c))).
```

This is logically equivalent to

```
(IMPLIES (AND  a  b)
         c)  .
```

Thus, we can assume a (the long formula) and b ($\neg$ I > 52) in the proof of c. All conjuncts of c, except the last one ((FORALL Y ...)), follow direcly from identical conjuncts in the formula a. The important subformula that must be proved is

```
(FORALL Y (IMPLIES (AND (LTQ 1 Y)
                        (LTQ Y (SUBTRACT (PLUS I 1) 1)))
                   (EQ (IF (EQ Y I)
                           (SELECT HOURS-WORKED-WEEKLY-IN
                                   (Y))
                           (SELECT HOURS-WORKED-WEEKLY-OUT
                                   (Y)))
                       (SELECT HOURS-WORKED-WEEKLY-IN (Y))))).
```

This formula incorporates the semantics of array assignment. The conditional expression (IF) indicates the place where array assignment has taken place. To prove this condition, we break the formula into two cases (written in infix form):

```
∀Y(1 ≤ Y ≤ I - 1 ⊃ HOURS-WORKED-WEEKLY-OUT(Y) =
                   HOURS-WORKED-WEEKLY-IN(Y))
∧ HOURS-WORKED-WEEKLY-IN(I) = HOURS-WORKED-WEEKLY-IN(I) .
```

The first case follows from formula a; the second case is an identity. Thus, the verification condition is proved.

The other verification conditions can be similarly proved, but it would be tedious to do so here.

F. Conclusions

We have shown how the proof of a simple COBOL program can expand into an enormous mass of material (e.g., the verification conditions are many times the length of the program itself). Fortunately, much of the volume can be reduced by simplification at various stages of the proof process. All the verification conditions--although long--are elementary proofs.

The simplicity of most proofs, together with the sheer number to be performed, are fundamental reasons why major research should be continued in semi-automatic deductive systems. Simplification packages and interactive programs to help the programmer direct the machine to a proof should also be part of this research in deductive systems.

Table VIII-1

EXAMPLE PROGRAM

```
DATA DIVISION.

FILE SECTION.

FD INPUT-MASTER-FILE.
01  INPUT-MASTER-RECORD.
    02  PERMANENT-INFORMATION.
        03  NAME-IN PICTURE X(35).
        03  SOCIAL-SECURITY-IN PICTURE 9(9).
        03  WEEKLY-SALARY-IN PICTURE 999V99.
    02  VARYING-INFORMATION.
        03  GROSS-PAY-TO-DATE-IN PICTURE 99999V99.
        03  HOURS-WORKED-TO-DATE-IN PICTURE 9999.
        03  CURRENT-WEEK-IN PICTURE 99.
    02  ARRAY-INFORMATION.
        03  HOURS-WORKED-WEEKLY-IN PICTURE 99 OCCURS 52 TIMES.

FD OUTPUT-MASTER-FILE.
01  OUTPUT-MASTER-RECORD.
    02  PERMANENT-INFORMATION.
        03  NAME-OUT PICTURE X(35).
        03  SOCIAL-SECURITY-OUT PICTURE 9(9).
        03  WEEKLY-SALARY-OUT PICTURE 999V99.
    02  VARYING-INFORMATION.
        03  GROSS-PAY-TO-DATE-OUT PICTURE 99999V99.
        03  HOURS-WORKED-TO-DATE-OUT PICTURE 9999.
        03  CURRENT-WEEK-OUT PICTURE 99.
    02  ARRAY-INFORMATION.
        03  HOURS-WORKED-WEEKLY-OUT PICTURE 99 OCCURS 52 TIMES.

FD TIME-CARD-FILE.
01  TIME-CARD.
    02  NAME PICTURE X(35).
    02  HOURS-WORKED-THIS-WEEK PICTURE 99.

FD PAYCHECK-FILE.
01  PAYCHECK.
    02  NAME PICTURE X(35).
    02  AMOUNT PICTURE 999V99.

WORKING-STORAGE SECTION.

77  THIS-WEEKS-PAY PICTURE 999V99.
77  I PICTURE 99.
77  FILE-FLAG PICTURE 9.

PROCEDURE DIVISION.
```

```
MAIN SECTION.

OPEN-FILES.
          (ASSERT 1).
          OPEN INPUT INPUT-MASTER-FILE.
          OPEN INPUT TIME-CARD-FILE.
          OPEN OUTPUT OUTPUT-MASTER-FILE.
          OPEN OUTPUT PAYCHECK-FILE.

LOOP.
          (ASSERT 2)
          PERFORM READ-INPUT-MASTER THRU READ-TIME-CARD.
          IF FILE-FLAG = 2 GO TO CLEANUP.
          IF FILE-FLAG NOT = 0 GO TO ERROR-ABORT.
          PERFORM PROCESS-RECORDS.
          PERFORM WRITE-OUTPUT.
          GO TO LOOP.

READ-INPUT-MASTER.
          READ INPUT-MASTER-FILE AT END ADD 1 TO FILE-FLAG.

READ-TIME-CARD.
          READ TIME-CARD-FILE AT END ADD 1 TO FILE-FLAG.

WRITE-OUTPUT.
          WRITE OUTPUT-MASTER-RECORD.
          WRITE PAYCHECK.

CLEANUP.
          (ASSERT 3).
          CLOSE INPUT-MASTER-FILE.
          CLOSE OUTPUT-MASTER-FILE.
          CLOSE PAYCHECK-FILE.
          STOP RUN (ASSERT 3).

ERROR-ABORT.
          (ASSERT 4).
          STOP RUN (ASSERT 4).

PROCESSING SECTION.

PROCESS-RECORDS.
          IF NAME-IN NOT = NAME OF TIME-CARD GO TO ERROR-ABORT.
          MOVE NAME-IN TO NAME-OUT.
          MOVE SOCIAL-SECURITY-IN TO SOCIAL-SECURITY-OUT.
          MOVE WEEKLY-SALARY-IN TO WEEKLY-SALARY-OUT.
          COMPUTE THIS-WEEKS-PAY ROUNDED = WEEKLY-SALARY-IN *
                  (HOURS-WORKED-THIS-WEEK / 40)
                  ON SIZE ERROR GO TO ERROR-ABORT.
          COMPUTE GROSS-PAY-TO-DATE-OUT = GROSS-PAY-TO-DATE-IN +
                  THIS-WEEKS-PAY.
          COMPUTE HOURS-WORKED-TO-DATE-OUT = HOURS-WORKED-TO-DATE-IN +
                  HOURS-WORKED-THIS-WEEK.
          COMPUTE CURRENT-WEEK-OUT = CURRENT-WEEK-IN + 1.
          PERFORM MOVE-ARRAY VARYING I FROM 1 BY 1 UNTIL I > 52
                 (ASSERT 7).
          COMPUTE HOURS-WORKED-WEEKLY-OUT (CURRENT-WEEK-OUT) =
                  HOURS-WORKED-THIS-WEEK.
          MOVE CORRESPONDING TIME-CARD TO PAYCHECK.
          MOVE THIS-WEEKS-PAY TO AMOUNT.

MOVE-ARRAY.
          MOVE HOURS-WORKED-WEEKLY-IN (I) TO
                  HOURS-WORKED-WEEKLY-OUT (I).

ERROR-ABORT.
          (ASSERT 5).
          STOP RUN (ASSERT 5).
```

Table VIII-2

ASSERTIONS FOR SAMPLE PROGRAM

```
(1 (AND [FORALL X (IMPLIES (AND (LTQ 1 X)
                                (LTQ X INPUT-MASTER-FILE.LENGTH))
                           (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                    (SELECT (QUAL NAME.ARRAY
                                                  TIME-CARD)
                                            (X)))
                                (GTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                               (X)))
                                (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                             (X))
                                     51]
        (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)))
(2 (AND (ASSERT 7)
        (LTQ INPUT-MASTER-FILE.INDEX INPUT-MASTER-FILE.LENGTH)
        (EQ FILE-FLAG 0)
        (EQ INPUT-MASTER-FILE.INDEX TIME-CARD-FILE.INDEX
            OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)))
(3 (AND (ASSERT 7)
        (GT INPUT-MASTER-FILE.INDEX INPUT-MASTER-FILE.LENGTH)
        (GT TIME-CARD-FILE.INDEX TIME-CARD-FILE.LENGTH)
        (EQ FILE-FLAG 2)
        (EQ INPUT-MASTER-FILE.LENGTH OUTPUT-MASTER-FILE.LENGTH
            PAYCHECK-FILE.LENGTH)))
(4 FALSE)
[5 (EXISTS X (AND (EQ X INPUT-MASTER-FILE.INDEX)
                  (OR (GT (TIMES (SELECT WEEKLY-SALARY-IN.ARRAY
                                         (X))
                                 (DIVIDE (SELECT
                                           HOURS-WORKED-THIS-WEEK.ARRAY
                                                   (X))
                                         40))
                          999.999)
                      (NEQ (SELECT NAME-IN.ARRAY (X))
                           (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                   (X]
[6 (AND (ASSERT 7)
        (EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
            (SUBTRACT TIME-CARD-FILE.INDEX 1)
            OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
        (EQ NAME-IN (QUAL NAME TIME-CARD)
            NAME-OUT
            (QUAL NAME PAYCHECK))
        [EQ GROSS-PAY-TO-DATE-OUT
            (PLUS GROSS-PAY-TO-DATE-IN
                  (ROUND 999V99 (TIMES WEEKLY-SALARY-IN
                                       (DIVIDE HOURS-WORKED-THIS-WEEK
                                               40]
        (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                                HOURS-WORKED-THIS-WEEK))
                (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
                (LTQ 1 I)
                (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                                        (LTQ Y (SUBTRACT I 1)))
                                   (EQ (SELECT HOURS-WORKED-WEEKLY-OUT
                                               (Y))
                                       (SELECT HOURS-WORKED-WEEKLY-IN
                                               (Y]
        [7
          (AND
            (ASSERT 1)
            (FORALL
              X
              (IMPLIES
                (AND (GTQ 1 X)
                     (LTQ X OUTPUT-MASTER-FILE.INDEX))
                (AND (EQ (SELECT NAME-IN.ARRAY (X))
                         (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                 (X))
                         (SELECT NAME-OUT.ARRAY (X))
                         (SELECT (QUAL NAME.ARRAY PAYCHECK)
                                 (X)))
                     [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
                         (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                               (ROUND 999V99 (TIMES (SELECT
                                                     WEEKLY-SALARY-IN.ARRAY
                                                              (X))
                                                     (DIVIDE (SELECT
                                              HOURS-WORKED-THIS-WEEK.ARRAY
                                                                      (X))
                                                              40]
                     [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                         (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                                       (X))
                               (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
                     [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                         (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
                     [FORALL Y (IMPLIES [AND (LTQ 1 Y)
                                             (LTQ Y 52)
                                             (NEQ Y (SELECT
                                                     CURRENT-WEEK-OUT.ARRAY
                                                              (X]
                                        (EQ (SELECT (SELECT
                                             HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                              (X))
                                                     (Y))
                                            (SELECT (SELECT
                                              HOURS-WORKED-WEEKLY-IN.ARRAY
                                                              (X))
                                                     (Y]
                     (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                         (X))
                (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
         (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
```

Table VIII-3

TRANSDUCED EXAMPLE PROGRAM (PROCEDURE DIVISION)

```
(PROCEDUREDIVISION$
 [SECTION$ MAIN (PARAGRAPH$ OPEN-FILES (ASSERT 1)
                            (OPENINPUT$ INPUT-MASTER-FILE)
                            (OPENINPUT$ TIME-CARD-FILE)
                            (OPENOUTPUT$ OUTPUT-MASTER-FILE)
                            (OPENOUTPUT$ PAYCHECK-FILE))
           (PARAGRAPH$ LOOP (ASSERT 2)
                       (PERFORM (ONCE$)
                                (DO$ READ-INPUT-MASTER READ-TIME-CARD)
                                NIL NIL)
                       (IF (EQ FILE-FLAG 2)
                           (GO CLEANUP)
                           (NEXT))
                       (IF (NEQ FILE-FLAG 0)
                           (GO ERROR-ABORT)
                           (NEXT))
                       (PERFORM (ONCE$)
                                (DO$ PROCESS-RECORDS PROCESS-RECORDS)
                                NIL NIL)
                       (PERFORM (ONCE$)
                                (DO$ WRITE-OUTPUT WRITE-OUTPUT)
                                NIL NIL)
                       (GO LOOP))
           (PARAGRAPH$ READ-INPUT-MASTER
                       (READ INPUT-MASTER-FILE NIL
                             (SET$ FILE-FLAG (PLUS FILE-FLAG 1)
                                   NIL)))
           (PARAGRAPH$ READ-TIME-CARD (READ TIME-CARD-FILE NIL
                                            (SET$ FILE-FLAG
                                                  (PLUS FILE-FLAG 1)
                                                  NIL)))
           (PARAGRAPH$ WRITE-OUTPUT (WRITE OUTPUT-MASTER-RECORD NIL)
                       (WRITE PAYCHECK NIL))
           (PARAGRAPH$ CLEANUP (ASSERT 3)
                       (CLOSE INPUT-MASTER-FILE)
                       (CLOSE TIME-CARD-FILE)
                       (CLOSE OUTPUT-MASTER-FILE)
                       (CLOSE PAYCHECK-FILE)
                       (STOP (ASSERT 3)))
           (PARAGRAPH$ ERROR-ABORT (ASSERT 4)
                       (STOP (ASSERT 4]
 [SECTION$ PROCESSING (PARAGRAPH$ PROCESS-RECORDS
                                  (IF (NEQ NAME-IN (QUAL NAME
                                                         TIME-CARD))
                                      (GO ERROR-ABORT)
                                      (NEXT))
                                  (SET$ NAME-OUT NAME-IN NIL)
                                  (SET$ SOCIAL-SECURITY-OUT
                                          SOCIAL-SECURITY-IN NIL)
                                    (SET$ WEEKLY-SALARY-OUT
                                          WEEKLY-SALARY-IN NIL)
                                    (SETROUNDED$ THIS-WEEKS-PAY
                                                 (TIMES WEEKLY-SALARY-IN
                                                        (DIVIDE
                                              HOURS-WORKED-THIS-WEEK 40))
                                                 (GO ERROR-ABORT))
                                    (SET$ GROSS-PAY-TO-DATE-OUT
                                          (PLUS GROSS-PAY-TO-DATE-IN
                                                THIS-WEEKS-PAY)
                                          NIL)
                                    (SET$ HOURS-WORKED-TO-DATE-OUT
                                          (PLUS HOURS-WORKED-TO-DATE-IN
                                              HOURS-WORKED-THIS-WEEK)
                                          NIL)
                                    (SET$ CURRENT-WEEK-OUT
                                          (PLUS 1 CURRENT-WEEK-IN)
                                          NIL)
                                    (PERFORM VARYING (DO$ MOVE-ARRAY
                                                          MOVE-ARRAY)
                                             (I 1 1 (GT I 52))
                                             (ASSERT 6))
                                    (SET$ (SELECT
                                             HOURS-WORKED-WEEKLY-OUT
                                                  (CURRENT-WEEK))
                                          HOURS-WORKED-THIS-WEEK NIL)
                                    (MOVECORRESPONDING$ TIME-CARD
                                                        PAYCHECK)
                                    (SET$ AMOUNT THIS-WEEKS-PAY NIL))
                 (PARAGRAPH$ MOVE-ARRAY (SET$ (SELECT
                                             HOURS-WORKED-WEEKLY-OUT
                                                  (I))
                                          (SELECT HOURS-WORKED-WEEKLY-IN
                                                  (I))
                                          NIL))
                 (PARAGRAPH$ ERROR-ABORT (ASSERT 5)
                             (STOP (ASSERT 5])
```

Table VIII-4

TRANSDUCED EXAMPLE PROGRAM (DATA DIVISION)

```
(DATADIVISION$
 [FILESECTION$ [FD INPUT-MASTER-FILE (((1 INPUT-MASTER-RECORD)
                     (((2 PERMANENT-INFORMATION)
                       ((3 NAME-IN X%(35%))
                        (3 SOCIAL-SECURITY-IN 9%(9%))
                        (3 WEEKLY-SALARY-IN 999V99)))
                      ((2 VARYING-INFORMATION)
                       ((3 GROSS-PAY-TO-DATE-IN 99999V99)
                        (3 HOURS-WORKED-TO-DATE-IN 9999)
                        (3 CURRENT-WEEK-IN 99)))
                      ((2 ARRAY-INFORMATION)
                       ((3 HOURS-WORKED-WEEKLY-IN 99 52]
               [FD OUTPUT-MASTER-FILE (((1 OUTPUT-MASTER-RECORD)
                     (((2 PERMANENT-INFORMATION)
                       ((3 NAME-OUT X%(35%))
                        (3 SOCIAL-SECURITY-OUT 9%(9%))
                        (3 WEEKLY-SALARY-OUT 999V99)))
                      ((2 VARYING-INFORMATION)
                       ((3 GROSS-PAY-TO-DATE-OUT 99999V99)
                        (3 HOURS-WORKED-TO-DATE 9999)
                        (3 CURRENT-WEEK-OUT 99)))
                      ((2 ARRAY-INFORMATION)
                       ((3 HOURS-WORKED-WEEKLY-OUT 99 52]
               [FD TIME-CARD-FILE (((1 TIME-CARD)
                      ((2 NAME X%(35%))
                       (2 HOURS-WORKED-THIS-WEEK 99]
               (FD PAYCHECK-FILE (((1 PAYCHECK)
                      ((2 NAME X%(35%))
                       (2 AMOUNT 999V99]
  (WORKINGSTORAGESECTION$ (77 THIS-WEEKS-PAY 999V99)
                          (77 I 999)
                          (77 FILE-FLAG 9)))
```

Table VIII-5

EXAMPLE PROGRAM AFTER POSTTRANSDUCTION PROCESSING
AND SIMPLIFICATION

```
((ASSERT 1)
 (ASSIGN INPUT-MASTER-FILE.INDEX 0)
 (ASSIGN TIME-CARD-FILE.INDEX 0)
 (ASSIGN OUTPUT-MASTER-FILE.INDEX 0)
 (ASSIGN PAYCHECK-FILE.INDEX 0)
 (ASSERT 2)
 [BLOCK (ASSIGN INPUT-MASTER-FILE.INDEX (PLUS INPUT-MASTER-FILE.INDEX
                                              1))
        (IF (GT INPUT-MASTER-FILE.INDEX INPUT-MASTER-FILE.LENGTH)
            (ASSIGN FILE-FLAG (PLUS FILE-FLAG 1))
            (NEXT))
        (ASSIGN NAME-IN (SELECT NAME-IN.ARRAY (INPUT-MASTER-FILE.INDEX)
                                ))
        (ASSIGN SOCIAL-SECURITY-IN (SELECT SOCIAL-SECURITY-IN.ARRAY
                                           (INPUT-MASTER-FILE.INDEX)))
        (ASSIGN WEEKLY-SALARY-IN (SELECT WEEKLY-SALARY-IN.ARRAY
                                         (INPUT-MASTER-FILE.INDEX)))
        (ASSIGN GROSS-PAY-TO-DATE-IN (SELECT
                                        GROSS-PAY-TO-DATE-IN.ARRAY
                                             (INPUT-MASTER-FILE.INDEX))
                )
        (ASSIGN HOURS-WORKED-TO-DATE-IN (SELECT
                                     HOURS-WORKED-TO-DATE-IN.ARRAY
                                                  (
INPUT-MASTER-FILE.INDEX)))
        (ASSIGN CURRENT-WEEK-IN (SELECT CURRENT-WEEK-IN.ARRAY
                                        (INPUT-MASTER-FILE.INDEX)))
        (ASSIGN HOURS-WORKED-WEEKLY-IN (SELECT
                                      HOURS-WORKED-WEEKLY-IN.ARRAY
                                               (INPUT-MASTER-FILE.INDEX
                                                 )))
        (ASSIGN TIME-CARD-FILE.INDEX (PLUS TIME-CARD-FILE.INDEX 1))
        (IF (GT TIME-CARD-FILE.INDEX TIME-CARD-FILE.LENGTH)
            (ASSIGN FILE-FLAG (PLUS FILE-FLAG 1))
            (NEXT))
        (ASSIGN (QUAL NAME TIME-CARD)
                (SELECT (QUAL NAME.ARRAY TIME-CARD)
                        (TIME-CARD-FILE.INDEX)))
        (ASSIGN HOURS-WORKED-THIS-WEEK
                (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (
                         TIME-CARD-FILE.INDEX]
 (IF (EQ FILE-FLAG 2)
     (GO (CLEANUP MAIN))
     (NEXT))
 (IF (NEQ FILE-FLAG 0)
     (GO (ERROR-ABORT MAIN))
     (NEXT))
```

```
(BLOCK (IF (NEQ NAME-IN (QUAL NAME TIME-CARD))
           (GO (ERROR-ABORT PROCESSING))
           (NEXT))
       (ASSIGN NAME-OUT NAME-IN)
       (ASSIGN SOCIAL-SECURITY-OUT SOCIAL-SECURITY-IN)
       (ASSIGN WEEKLY-SALARY-OUT WEEKLY-SALARY-IN)
       [IF (GT [ABS (ROUND THIS-WEEKS-PAY (TIMES WEEKLY-SALARY-IN
                                                 (DIVIDE
                                            HOURS-WORKED-THIS-WEEK 40]
               999.99)
           (GO (ERROR-ABORT PROCESSING))
           (ASSIGN THIS-WEEKS-PAY (ROUND THIS-WEEKS-PAY
                                          (TIMES WEEKLY-SALARY-IN
                                                 (DIVIDE
                                            HOURS-WORKED-THIS-WEEK 40]
       (ASSIGN GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                           THIS-WEEKS-PAY))
       (ASSIGN HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                            HOURS-WORKED-THIS-WEEK))
       (ASSIGN CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
       [BLOCK (ASSIGN I 1)
              (ASSERT 6)
              (IF (GT I 52)
                  (ENDPERFORM)
                  (NEXT))
              (ASSIGN (SELECT HOURS-WORKED-WEEKLY-OUT (I))
                      (SELECT HOURS-WORKED-WEEKLY-IN (I)))
              (ASSIGN I (PLUS I 1))
              (LOOPASSERT (ASSERT 6]
       (ASSIGN (SELECT HOURS-WORKED-WEEKLY-OUT (CURRENT-WEEK))
               HOURS-WORKED-THIS-WEEK)
       (ASSIGN (QUAL NAME PAYCHECK)
               (QUAL NAME TIME-CARD))
       (ASSIGN AMOUNT THIS-WEEKS-PAY))
(BLOCK (ASSIGN OUTPUT-MASTER-FILE.INDEX (PLUS
                                          OUTPUT-MASTER-FILE.INDEX 1))
       (ASSIGN OUTPUT-MASTER-FILE.LENGTH (PLUS
                                         OUTPUT-MASTER-FILE.LENGTH 1))
       (ASSIGN (SELECT NAME-OUT.ARRAY (OUTPUT-MASTER-FILE.INDEX))
               NAME-OUT)
       (ASSIGN (SELECT SOCIAL-SECURITY-OUT.ARRAY (
                         OUTPUT-MASTER-FILE.INDEX))
               SOCIAL-SECURITY-OUT)
       (ASSIGN (SELECT WEEKLY-SALARY-OUT.ARRAY (
                         OUTPUT-MASTER-FILE.INDEX))
               WEEKLY-SALARY-OUT)
       (ASSIGN (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (
                         OUTPUT-MASTER-FILE.INDEX))
               GROSS-PAY-TO-DATE-OUT)
       (ASSIGN (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (
                         OUTPUT-MASTER-FILE.INDEX))
               HOURS-WORKED-TO-DATE-OUT)
       (ASSIGN (SELECT CURRENT-WEEK-OUT.ARRAY (
                              OUTPUT-MASTER-FILE.INDEX))
                    CURRENT-WEEK-OUT)
          (ASSIGN (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY (
                              OUTPUT-MASTER-FILE.INDEX))
                    HOURS-WORKED-WEEKLY-OUT)
          (ASSIGN PAYCHECK-FILE.INDEX (PLUS PAYCHECK-FILE.INDEX 1))
          (ASSIGN PAYCHECK-FILE.LENGTH (PLUS PAYCHECK-FILE.LENGTH 1))
          (ASSIGN (SELECT (QUAL NAME.ARRAY PAYCHECK)
                            (PAYCHECK-FILE.INDEX))
                    (QUAL NAME PAYCHECK))
          (ASSIGN (SELECT AMOUNT.ARRAY (PAYCHECK-FILE.INDEX))
                    AMOUNT))
(GO (LOOP MAIN))
(ASSERT 3)
(STOP (ASSERT 3))
(ASSERT 4)
(STOP (ASSERT 4)))
```

Table VIII-6

SIGNIFICANT PATHS FOR THE EXAMPLE PROGRAM

```
((ASSERT 2)
 (ASSIGN PAYCHECK-FILE.INDEX 0)
 (ASSIGN OUTPUT-MASTER-FILE.INDEX 0)
 (ASSIGN TIME-CARD-FILE.INDEX 0)
 (ASSIGN INPUT-MASTER-FILE.INDEX 0)
 (ASSERT 1))
((ASSERT 3)
 (IF (EQ FILE-FLAG 2))
 (ASSIGN HOURS-WORKED-THIS-WEEK (SELECT HOURS-WORKED-THIS-WEEK.ARRAY
                                          (TIME-CARD-FILE.INDEX)))
 (ASSIGN (QUAL NAME TIME-CARD)
         (SELECT (QUAL NAME.ARRAY TIME-CARD)
                 (TIME-CARD-FILE.INDEX)))
 (ASSIGN FILE-FLAG (PLUS FILE-FLAG 1))
 (IF (GT TIME-CARD-FILE.INDEX TIME-CARD-FILE.LENGTH))
 (ASSIGN TIME-CARD-FILE.INDEX (PLUS TIME-CARD-FILE.INDEX 1))
 (ASSIGN HOURS-WORKED-WEEKLY-IN (SELECT HOURS-WORKED-WEEKLY-IN.ARRAY
                                          (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN CURRENT-WEEK-IN (SELECT CURRENT-WEEK-IN.ARRAY (
                                     INPUT-MASTER-FILE.INDEX)))
 (ASSIGN HOURS-WORKED-TO-DATE-IN (SELECT
                                      HOURS-WORKED-TO-DATE-IN.ARRAY
                                           (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN GROSS-PAY-TO-DATE-IN (SELECT GROSS-PAY-TO-DATE-IN.ARRAY
                                        (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN WEEKLY-SALARY-IN (SELECT WEEKLY-SALARY-IN.ARRAY (
                                      INPUT-MASTER-FILE.INDEX)))
 (ASSIGN SOCIAL-SECURITY-IN (SELECT SOCIAL-SECURITY-IN.ARRAY
                                      (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN NAME-IN (SELECT NAME-IN.ARRAY (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN FILE-FLAG (PLUS FILE-FLAG 1))
 (IF (GT INPUT-MASTER-FILE.INDEX INPUT-MASTER-FILE.LENGTH))
 (ASSIGN INPUT-MASTER-FILE.INDEX (PLUS INPUT-MASTER-FILE.INDEX 1))
 (ASSERT 2))
((ASSERT 5)
 (IF (GT [ABS (ROUND THIS-WEEKS-PAY (TIMES WEEKLY-SALARY-IN
                                            (DIVIDE
                                            HOURS-WORKED-THIS-WEEK 40]
         999.99))
 (ASSIGN WEEKLY-SALARY-OUT WEEKLY-SALARY-IN)
 (ASSIGN SOCIAL-SECURITY-OUT SOCIAL-SECURITY-IN)
 (ASSIGN NAME-OUT NAME-IN)
 [IF (NOT (NEQ NAME-IN (QUAL NAME TIME-CARD]
 (IF (NOT (NEQ FILE-FLAG 0)))
 (IF (NOT (EQ FILE-FLAG 2)))
 (ASSIGN HOURS-WORKED-THIS-WEEK (SELECT HOURS-WORKED-THIS-WEEK.ARRAY
                                          (TIME-CARD-FILE.INDEX)))
 (ASSIGN (QUAL NAME TIME-CARD)
        (SELECT (QUAL NAME.ARRAY TIME-CARD)
                (TIME-CARD-FILE.INDEX)))
 (IF (NOT (GT TIME-CARD-FILE.INDEX TIME-CARD-FILE.LENGTH)))
 (ASSIGN TIME-CARD-FILE.INDEX (PLUS TIME-CARD-FILE.INDEX 1))
 (ASSIGN HOURS-WORKED-WEEKLY-IN (SELECT HOURS-WORKED-WEEKLY-IN.ARRAY
                                    (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN CURRENT-WEEK-IN (SELECT CURRENT-WEEK-IN.ARRAY (
                               INPUT-MASTER-FILE.INDEX)))
 (ASSIGN HOURS-WORKED-TO-DATE-IN (SELECT
                                HOURS-WORKED-TO-DATE-IN.ARRAY
                                    (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN GROSS-PAY-TO-DATE-IN (SELECT GROSS-PAY-TO-DATE-IN.ARRAY
                                  (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN WEEKLY-SALARY-IN (SELECT WEEKLY-SALARY-IN.ARRAY (
                                INPUT-MASTER-FILE.INDEX)))
 (ASSIGN SOCIAL-SECURITY-IN (SELECT SOCIAL-SECURITY-IN.ARRAY
                                (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN NAME-IN (SELECT NAME-IN.ARRAY (INPUT-MASTER-FILE.INDEX)))
 (IF (NOT (GT INPUT-MASTER-FILE.INDEX INPUT-MASTER-FILE.LENGTH)))
 (ASSIGN INPUT-MASTER-FILE.INDEX (PLUS INPUT-MASTER-FILE.INDEX 1))
 (ASSERT 2))
((ASSERT 6)
 (ASSIGN I 1)
 (ASSIGN CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
 (ASSIGN HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                    HOURS-WORKED-THIS-WEEK))
 (ASSIGN GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                 THIS-WEEKS-PAY))
 [ASSIGN THIS-WEEKS-PAY (ROUND THIS-WEEKS-PAY
                          (TIMES WEEKLY-SALARY-IN
                                 (DIVIDE HOURS-WORKED-THIS-WEEK
                                         40]
 (IF (NOT (GT [ABS (ROUND THIS-WEEKS-PAY (TIMES WEEKLY-SALARY-IN
                                               (DIVIDE
                                          HOURS-WORKED-THIS-WEEK 40]
              999.99)))
 (ASSIGN WEEKLY-SALARY-OUT WEEKLY-SALARY-IN)
 (ASSIGN SOCIAL-SECURITY-OUT SOCIAL-SECURITY-IN)
 (ASSIGN NAME-OUT NAME-IN)
 [IF (NOT (NEQ NAME-IN (QUAL NAME TIME-CARD]
 (IF (NOT (NEQ FILE-FLAG 0)))
 (IF (NOT (EQ FILE-FLAG 2)))
 (ASSIGN HOURS-WORKED-THIS-WEEK (SELECT HOURS-WORKED-THIS-WEEK.ARRAY
                                    (TIME-CARD-FILE.INDEX)))
 (ASSIGN (QUAL NAME TIME-CARD)
        (SELECT (QUAL NAME.ARRAY TIME-CARD)
                (TIME-CARD-FILE.INDEX)))
 (IF (NOT (GT TIME-CARD-FILE.INDEX TIME-CARD-FILE.LENGTH)))
 (ASSIGN TIME-CARD-FILE.INDEX (PLUS TIME-CARD-FILE.INDEX 1))
 (ASSIGN HOURS-WORKED-WEEKLY-IN (SELECT HOURS-WORKED-WEEKLY-IN.ARRAY
                                    (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN CURRENT-WEEK-IN (SELECT CURRENT-WEEK-IN.ARRAY (
                               INPUT-MASTER-FILE.INDEX)))
 (ASSIGN HOURS-WORKED-TO-DATE-IN (SELECT
                                     HOURS-WORKED-TO-DATE-IN.ARRAY
                                          (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN GROSS-PAY-TO-DATE-IN (SELECT GROSS-PAY-TO-DATE-IN.ARRAY
                                       (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN WEEKLY-SALARY-IN (SELECT WEEKLY-SALARY-IN.ARRAY (
                                     INPUT-MASTER-FILE.INDEX)))
 (ASSIGN SOCIAL-SECURITY-IN (SELECT SOCIAL-SECURITY-IN.ARRAY
                                     (INPUT-MASTER-FILE.INDEX)))
 (ASSIGN NAME-IN (SELECT NAME-IN.ARRAY (INPUT-MASTER-FILE.INDEX)))
 (IF (NOT (GT INPUT-MASTER-FILE.INDEX INPUT-MASTER-FILE.LENGTH)))
 (ASSIGN INPUT-MASTER-FILE.INDEX (PLUS INPUT-MASTER-FILE.INDEX 1))
 (ASSERT 2))
((ASSERT 5)
 (ASSIGN I (PLUS I 1))
 (ASSIGN (SELECT HOURS-WORKED-WEEKLY-OUT (I))
         (SELECT HOURS-WORKED-WEEKLY-IN (I)))
 (IF (NOT (GT I 52)))
 (ASSERT 6))
((ASSERT 2)
 (ASSIGN (SELECT AMOUNT.ARRAY (PAYCHECK-FILE.INDEX))
         (QUAL AMOUNT PAYCHECK))
 (ASSIGN (SELECT (QUAL NAME.ARRAY PAYCHECK)
                 (PAYCHECK-FILE.INDEX))
         (QUAL NAME PAYCHECK))
 (ASSIGN PAYCHECK-FILE.LENGTH (PLUS PAYCHECK-FILE.LENGTH 1))
 (ASSIGN PAYCHECK-FILE.INDEX (PLUS PAYCHECK-FILE.INDEX 1))
 (ASSIGN (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY (
                 OUTPUT-MASTER-FILE.INDEX))
         HOURS-WORKED-WEEKLY-OUT)
 (ASSIGN (SELECT CURRENT-WEEK-OUT.ARRAY (OUTPUT-MASTER-FILE.INDEX))
         CURRENT-WEEK-OUT)
 (ASSIGN (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (
                 OUTPUT-MASTER-FILE.INDEX))
         HOURS-WORKED-TO-DATE-OUT)
 (ASSIGN (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (
                 OUTPUT-MASTER-FILE.INDEX))
         GROSS-PAY-TO-DATE-OUT)
 (ASSIGN (SELECT WEEKLY-SALARY-OUT.ARRAY (OUTPUT-MASTER-FILE.INDEX))
         WEEKLY-SALARY-OUT)
 (ASSIGN (SELECT SOCIAL-SECURITY-OUT.ARRAY (OUTPUT-MASTER-FILE.INDEX))
         SOCIAL-SECURITY-OUT)
 (ASSIGN (SELECT NAME-OUT.ARRAY (OUTPUT-MASTER-FILE.INDEX))
         NAME-OUT)
 (ASSIGN OUTPUT-MASTER-FILE.LENGTH (PLUS OUTPUT-MASTER-FILE.LENGTH 1))
 (ASSIGN OUTPUT-MASTER-FILE.INDEX (PLUS OUTPUT-MASTER-FILE.INDEX 1))
 (ASSIGN AMOUNT THIS-WEEKS-PAY)
 (ASSIGN (QUAL NAME PAYCHECK)
         (QUAL NAME TIME-CARD))
 (ASSIGN (SELECT HOURS-WORKED-WEEKLY-OUT (CURRENT-WEEK))
         HOURS-WORKED-THIS-WEEK)
 (IF (GT I 52))
 (ASSERT 6))
```

Table VIII-7

VERIFICATION CONDITION FOR PATH (6-6) OF EXAMPLE PROGRAM

```
(IMPLIES
 [AND
   [FORALL X (IMPLIES (AND (LTQ 1 X)
                           (LTQ X INPUT-MASTER-FILE.LENGTH))
                      (AND (EQ (SELECT NAME-IN.ARRAY (X))
                               (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                       (X)))
                           (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                          (X)))
                           (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                        (X))
                                51]
   (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
   [FORALL
     X
     (IMPLIES
       (AND (LTQ 1 X)
            (LTQ X OUTPUT-MASTER-FILE.INDEX))
       (AND (EQ (SELECT NAME-IN.ARRAY (X))
                (SELECT (QUAL NAME.ARRAY TIME-CARD)
                        (X))
                (SELECT NAME-OUT.ARRAY (X))
                (SELECT (QUAL NAME.ARRAY PAYCHECK)
                        (X)))
            [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
                (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                      (ROUND 999V99 (TIMES (SELECT
                                              WEEKLY-SALARY-IN.ARRAY
                                                     (X))
                                             (DIVIDE (SELECT
                                        HOURS-WORKED-THIS-WEEK.ARRAY
                                                             (X))
                                                     40]
            [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY (X))
                      (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
            [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
            [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                    (LTQ Y 52)
                                    (NEQ Y (SELECT
                                              CURRENT-WEEK-OUT.ARRAY
                                                     (X]
                               (EQ (SELECT (SELECT
                                     HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                     (X))
                                           (Y))
                                   (SELECT (SELECT
                                          HOURS-WORKED-WEEKLY-IN.ARRAY
                                                   (X))
                                              (Y]
                   (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                       (X))
                               (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
                         (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
      (EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
          (SUBTRACT TIME-CARD-FILE.INDEX 1)
          OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
      (EQ NAME-IN (QUAL NAME TIME-CARD)
          NAME-OUT
          (QUAL NAME PAYCHECK))
      [EQ GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                       (ROUND 999V99
                                              (TIMES WEEKLY-SALARY-IN
                                                     (DIVIDE
                                                HOURS-WORKED-THIS-WEEK 40]
      (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                         HOURS-WORKED-THIS-WEEK))
      (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
      (LTQ 1 I)
      (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                              (LTQ Y (SUBTRACT I 1)))
                         (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
                             (SELECT HOURS-WORKED-WEEKLY-IN (Y]
    [IMPLIES
      (NOT (GT I 52))
      (AND
        [FORALL X (IMPLIES (AND (LTQ 1 X)
                                (LTQ X INPUT-MASTER-FILE.LENGTH))
                           (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                    (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                            (X)))
                                (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                               (X)))
                                (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                             (X))
                                     51]
        (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
        [FORALL
          X
          (IMPLIES
            (AND (LTQ 1 X)
                 (LTQ X OUTPUT-MASTER-FILE.INDEX))
            (AND (EQ (SELECT NAME-IN.ARRAY (X))
                     (SELECT (QUAL NAME.ARRAY TIME-CARD)
                             (X))
                     (SELECT NAME-OUT.ARRAY (X))
                     (SELECT (QUAL NAME.ARRAY PAYCHECK)
                             (X)))
                 [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
                     (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                                  (ROUND 999V99 (TIMES (SELECT
                                                 WEEKLY-SALARY-IN.ARRAY
                                                        (X))
                                                  (DIVIDE (SELECT
                                            HOURS-WORKED-THIS-WEEK.ARRAY
                                                                (X))
                                                        40]
                         [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                             (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                                          (X))
                                  (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
                         [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                             (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
                         [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                            (LTQ Y 52)
                                            (NEQ Y (SELECT
                                                 CURRENT-WEEK-OUT.ARRAY
                                                        (X]
                                        (EQ (SELECT (SELECT
                                           HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                        (X))
                                                  (Y))
                                           (SELECT (SELECT
                                            HOURS-WORKED-WEEKLY-IN.ARRAY
                                                        (X))
                                                  (Y]
                         (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                            (X))
                                    (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
                             (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
                  (EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
                     (SUBTRACT TIME-CARD-FILE.INDEX 1)
                     OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
                  (EQ NAME-IN (QUAL NAME TIME-CARD)
                     NAME-OUT
                     (QUAL NAME PAYCHECK))
                  [EQ GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                           (ROUND 999V99
                                                 (TIMES WEEKLY-SALARY-IN
                                                       (DIVIDE
                                                 HOURS-WORKED-THIS-WEEK 40]
                  (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                              HOURS-WORKED-THIS-WEEK))
                  (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
                  (LTQ 1 I)
                  (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                                         (LTQ Y I))
                                  (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                                     (IF (EQ Y I)
                                        (SELECT HOURS-WORKED-WEEKLY-IN
                                              (Y))
                                        (SELECT HOURS-WORKED-WEEKLY-OUT
                                              (Y]))
```

Table VIII-8

PROOF OF VERIFICATION CONDITION (6-5) OF EXAMPLE PROGRAM

```
[1
  (AND
    [FORALL X (IMPLIES (AND (LTQ 1 X)
                            (LTQ X INPUT-MASTER-FILE.LENGTH))
                       (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                        (X)))
                            (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                           (X)))
                            (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                         (X))
                                 51]
    (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
    [FORALL
      X
      (IMPLIES
        (AND (LTQ 1 X)
             (LTQ X OUTPUT-MASTER-FILE.INDEX))
        (AND (EQ (SELECT NAME-IN.ARRAY (X))
                 (SELECT (QUAL NAME.ARRAY TIME-CARD)
                         (X))
                 (SELECT NAME-OUT.ARRAY (X))
                 (SELECT (QUAL NAME.ARRAY PAYCHECK)
                         (X)))
             [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
                 (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                       (ROUND 999V99 (TIMES (SELECT
                                              WEEKLY-SALARY-IN.ARRAY
                                                     (X))
                                              (DIVIDE (SELECT
                                        HOURS-WORKED-THIS-WEEK.ARRAY
                                                             (X))
                                                     40]
             [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                 (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                               (X))
                       (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
             [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                 (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
             [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                     (LTQ Y 52)
                                     (NEQ Y (SELECT
                                              CURRENT-WEEK-OUT.ARRAY
                                                     (X]
                                (EQ (SELECT (SELECT
                                      HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                     (X))
                                              (Y))
```

```
                                    (SELECT (SELECT
                                     HOURS-WORKED-WEEKLY-IN.ARRAY
                                                    (X))
                                            (Y]
             (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                 (X))
                         (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
                 (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
    (EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
        (SUBTRACT TIME-CARD-FILE.INDEX 1)
        OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
    (EQ NAME-IN (QUAL NAME TIME-CARD)
        NAME-OUT
        (QUAL NAME PAYCHECK))
    [EQ GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                    (ROUND 999V99
                                           (TIMES WEEKLY-SALARY-IN
                                                  (DIVIDE
                                           HOURS-WORKED-THIS-WEEK 40]
    (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                       HOURS-WORKED-THIS-WEEK))
    (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
    (LTQ 1 I)
    (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                            (LTQ Y (SUBTRACT I 1)))
                       (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
                           (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[2 (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                           (LTQ Y (SUBTRACT I 1)))
                      (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
                          (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[3 (IMPLIES (AND (LTQ 1 Y)
                 (LTQ Y (SUBTRACT I 1)))
            (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
                (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[4 (AND (LTQ 1 Y)
        (LTQ Y (SUBTRACT I 1]
[5 (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
       (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[6 (IMPLIES (NOT (EQ Y I))
            (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
                (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[7 (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
      (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[8 (IMPLIES (EQ Y I)
            (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[9 (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
       (IF (EQ Y I)
           (SELECT HOURS-WORKED-WEEKLY-IN (Y))
           (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
[10 (IMPLIES (AND (LTQ 1 Y)
                  (LTQ Y (SUBTRACT I 1)))
                 (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                     (IF (EQ Y I)
                         (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                         (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
(11 (EQ Y I))
(12 (NOT (EQ Y I)))
[13 (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
        (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[14 (IMPLIES (NOT (EQ Y I))
             (EQ (SELECT HOURS-WORKED-WEEKLY-OUT (Y))
                 (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[15 (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
        (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[16 (IMPLIES (EQ Y I)
             (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                 (SELECT HOURS-WORKED-WEEKLY-IN (Y]
[17 (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
        (IF (EQ Y I)
            (SELECT HOURS-WORKED-WEEKLY-IN (Y))
            (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
[18 (IMPLIES (EQ Y I)
             (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                 (IF (EQ Y I)
                     (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                     (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
[19 (IMPLIES (OR (AND (LTQ 1 Y)
                      (LTQ Y (SUBTRACT I 1)))
                 (EQ Y I))
             (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                 (IF (EQ Y I)
                     (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                     (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
[20 (AND (LTQ 1 Y)
         (LTQ Y (SUBTRACT I 1]
(21 (LTQ Y I))
(22 (OR (LTQ Y (SUBTRACT I 1))
        (EQ Y I)))
(23 (LTQ Y (SUBTRACT I 1)))
(24 (LTQ 1 Y))
[25 (AND (LTQ 1 Y)
         (LTQ Y (SUBTRACT I 1]
(26 (OR (AND (LTQ 1 Y)
             (LTQ Y (SUBTRACT I 1)))
        (EQ Y I)))
[27 (IMPLIES (LTQ Y (SUBTRACT I 1))
             (OR (AND (LTQ 1 Y)
                      (LTQ Y (SUBTRACT I 1)))
                 (EQ Y I]
(28 (EQ Y I))
(29 (OR (AND (LTQ 1 Y)
             (LTQ Y (SUBTRACT I 1)))
        (EQ Y I)))
[30 (IMPLIES (EQ Y I)
           (OR (AND (LTQ 1 Y)
                    (LTQ Y (SUBTRACT I 1)))
               (EQ Y I]
L31 (OR (AND (LTQ 1 Y)
           (LTQ Y (SUBTRACT I 1)))
       (EQ Y I)))
L32 (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
       (IF (EQ Y I)
          (SELECT HOURS-WORKED-WEEKLY-IN (Y))
          (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
L33 (IMPLIES (AND (LTQ 1 Y)
                (LTQ Y I))
           (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
               (IF (EQ Y I)
                   (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                   (SELECT HOURS-WORKED-WEEKLY-OUT (Y]
L34 (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                        (LTQ Y I))
                    (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                       (IF (EQ Y I)
                           (SELECT HOURS-WORKED-WEEKLY-IN
                                 (Y))
                           (SELECT HOURS-WORKED-WEEKLY-OUT
                                 (Y]

L35
  (AND
    (FORALL X (IMPLIES (AND (LTQ 1 X)
                        (LTQ X INPUT-MASTER-FILE.LENGTH))
                     AND (EQ (SELECT NAME-IN.ARRAY (X))
                            (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                  (X)))
                        (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                     (X)))
                        (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                   (X))
                             51)
    (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
    (FORALL
      X
      (IMPLIES
        (AND (LTQ 1 X
            (LTQ X OUTPUT-MASTER-FILE.INDEX))
        (AND (EQ (SELECT NAME-IN.ARRAY (X))
               (SELECT (QUAL NAME.ARRAY TIME-CARD)
                      (X))
               (SELECT NAME-OUT.ARRAY (X))
               (SELECT (QUAL NAME.ARRAY PAYCHECK)
                      (X)))
            (EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
               (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                    (ROUND 99V99 (TIMES (SELECT
                                     WEEKLY-SALARY-IN.ARRAY
                                           (X))
```

```
                                            (DIVIDE (SELECT
                                     HOURS-WORKED-THIS-WEEK.ARRAY
                                                            (X))
                                                    40]
             [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                 (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                               (X))
                       (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
             [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                 (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
             [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                     (LTQ Y 52)
                                     (NEQ Y (SELECT
                                           CURRENT-WEEK-OUT.ARRAY
                                                    (X]
                                (EQ (SELECT (SELECT
                                    HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                    (X))
                                            (Y))
                                    (SELECT (SELECT
                                     HOURS-WORKED-WEEKLY-IN.ARRAY
                                                    (X))
                                            (Y]
             (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                 (X))
                         (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
                 (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
    (EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
        (SUBTRACT TIME-CARD-FILE.INDEX 1)
        OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
    (EQ NAME-IN (QUAL NAME TIME-CARD)
        NAME-OUT
        (QUAL NAME PAYCHECK))
    [EQ GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                    (ROUND 999V99
                                           (TIMES WEEKLY-SALARY-IN
                                                  (DIVIDE
                                           HOURS-WORKED-THIS-WEEK 40]
    (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                       HOURS-WORKED-THIS-WEEK))
    (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
    (LTQ 1 I)))
[36
  (AND
    [FORALL X (IMPLIES (AND (LTQ 1 X)
                            (LTQ X INPUT-MASTER-FILE.LENGTH))
                       (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                        (X)))
                            (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                           (X)))
                            (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                         (X))
```

```
                             51]
(EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
[FORALL
  X
  (IMPLIES
    (AND (LTQ 1 X)
         (LTQ X OUTPUT-MASTER-FILE.INDEX))
    (AND (EQ (SELECT NAME-IN.ARRAY (X))
             (SELECT (QUAL NAME.ARRAY TIME-CARD)
                     (X))
             (SELECT NAME-OUT.ARRAY (X))
             (SELECT (QUAL NAME.ARRAY PAYCHECK)
                     (X)))
         (EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
             (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                   (ROUND 999V99 (TIMES (SELECT
                                       WEEKLY-SALARY-IN.ARRAY
                                                (X))
                                        (DIVIDE (SELECT
                                 HOURS-WORKED-THIS-WEEK.ARRAY
                                                        (X))
                                              40]
         (EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
             (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                           (X))
                   (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
         (EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
             (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
         [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                 (LTQ Y 52)
                                 (NEQ Y (SELECT
                                       CURRENT-WEEK-OUT.ARRAY
                                                (X]
                            (EQ (SELECT (SELECT
                                HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                (X))
                                        (Y))
                                (SELECT (SELECT
                                 HOURS-WORKED-WEEKLY-IN.ARRAY
                                                (X))
                                        (Y]
         (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                             (X))
                     (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
             (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
(EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
    (SUBTRACT TIME-CARD-FILE.INDEX 1)
    OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
(EQ NAME-IN (QUAL NAME TIME-CARD)
    NAME-OUT
    (QUAL NAME PAYCHECK))
(EQ GROSS-PAY-TO-DATE-OUT (PLUS GROSS-PAY-TO-DATE-IN
                                (ROUND 999V99
                                                    (TIMES WEEKLY-SALARY-IN
                                                           (DIVIDE
                                                    HOURS-WORKED-THIS-WEEK 40]
            (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                               HOURS-WORKED-THIS-WEEK))
            (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
            (LTQ 1 I)
            (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                                    (LTQ Y I))
                               (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                                   (IF (EQ Y I)
                                       (SELECT HOURS-WORKED-WEEKLY-IN
                                               (Y))
                                       (SELECT HOURS-WORKED-WEEKLY-OUT
                                               (Y]
L37
  (IMPLIES
    (NOT (GT I 52))
    (AND
      [FORALL X (IMPLIES (AND (LTQ 1 X)
                              (LTQ X INPUT-MASTER-FILE.LENGTH))
                         (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                  (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                          (X)))
                              (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                             (X)))
                              (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                           (X))
                                   51]
      (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
      [FORALL
        X
        (IMPLIES
          (AND (LTQ 1 X)
               (LTQ X OUTPUT-MASTER-FILE.INDEX))
          (AND (EQ (SELECT NAME-IN.ARRAY (X))
                   (SELECT (QUAL NAME.ARRAY TIME-CARD)
                           (X))
                   (SELECT NAME-OUT.ARRAY (X))
                   (SELECT (QUAL NAME.ARRAY PAYCHECK)
                           (X)))
               [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
                   (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                         (ROUND 999V99 (TIMES (SELECT
                                                WEEKLY-SALARY-IN.ARRAY
                                                           (X))
                                                 (DIVIDE
                                                   (SELECT
                                          HOURS-WORKED-THIS-WEEK.ARRAY
                                                             (X))
                                                   40]
               [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                   (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                                                (X))
                                        (SELECT HOURS-WORKED-THIS-WEEK.ARRAY
                                                (X]
                              [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                                  (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
                              [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                                     (LTQ Y 52)
                                                     (NEQ Y (SELECT
                                                         CURRENT-WEEK-OUT.ARRAY
                                                                    (X]
                                                (EQ (SELECT (SELECT
                                                  HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                                    (X))
                                                            (Y))
                                                    (SELECT (SELECT
                                                     HOURS-WORKED-WEEKLY-IN.ARRAY
                                                                    (X))
                                                            (Y]
                              (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                  (X))
                                          (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
                                  (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
                      (EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
                          (SUBTRACT TIME-CARD-FILE.INDEX 1)
                          OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
                      (EQ NAME-IN (QUAL NAME TIME-CARD)
                          NAME-OUT
                          (QUAL NAME PAYCHECK))
                      [EQ GROSS-PAY-TO-DATE-OUT
                          (PLUS GROSS-PAY-TO-DATE-IN (ROUND 999V99
                                                        (TIMES WEEKLY-SALARY-IN
                                                               (DIVIDE
                                                       HOURS-WORKED-THIS-WEEK 40]
                      (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                                         HOURS-WORKED-THIS-WEEK))
                      (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
                      (LTQ 1 I)
                      (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                                              (LTQ Y I))
                                         (EQ (SELECT HOURS-WORKED-WEEKLY-IN (Y))
                                             (IF (EQ Y I)
                                                 (SELECT HOURS-WORKED-WEEKLY-IN
                                                         (Y))
                                                 (SELECT HOURS-WORKED-WEEKLY-OUT
                                                         (Y]
                [38
                  (IMPLIES
                    [AND
                      [FORALL X (IMPLIES (AND (LTQ 1 X)
                                              (LTQ X INPUT-MASTER-FILE.LENGTH))
                                         (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                                  (SELECT (QUAL NAME.ARRAY TIME-CARD)
                                                          (X)))
```

BEST AVAILABLE COPY

```
                         (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                        (X)))
                         (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                      (X))
                              51]
(EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
[FORALL
  X
  (IMPLIES
    (AND (LTQ 1 X)
         (LTQ X OUTPUT-MASTER-FILE.INDEX))
    (AND (EQ (SELECT NAME-IN.ARRAY (X))
             (SELECT (QUAL NAME.ARRAY TIME-CARD)
                     (X))
             (SELECT NAME-OUT.ARRAY (X))
             (SELECT (QUAL NAME.ARRAY PAYCHECK)
                     (X)))
         [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
             (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY (X))
                   (ROUND 999V99 (TIMES (SELECT
                                         WEEKLY-SALARY-IN.ARRAY
                                               (X))
                                         (DIVIDE
                                          (SELECT
                              HOURS-WORKED-THIS-WEEK.ARRAY
                                                (X))
                                          40]
         [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
             (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                           (X))
                   (SELECT HOURS-WORKED-THIS-WEEK.ARRAY
                           (X]
         [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
             (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
         [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                 (LTQ Y 52)
                                 (NEQ Y (SELECT
                                         CURRENT-WEEK-OUT.ARRAY
                                               (X]
                            (EQ (SELECT (SELECT
                              HOURS-WORKED-WEEKLY-OUT.ARRAY
                                               (X))
                                        (Y))
                                (SELECT (SELECT
                              HOURS-WORKED-WEEKLY-IN.ARRAY
                                               (X))
                                        (Y]
         (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                             (X))
                     (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
             (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
(EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
    (SUBTRACT TIME-CARD-FILE.INDEX 1)
```

```
        OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
    (EQ NAME-IN (QUAL NAME TIME-CARD)
        NAME-OUT
        (QUAL NAME PAYCHECK))
    [EQ GROSS-PAY-TO-DATE-OUT
        (PLUS GROSS-PAY-TO-DATE-IN (ROUND 999V99
                                          (TIMES WEEKLY-SALARY-IN
                                                 (DIVIDE
                                         HOURS-WORKED-THIS-WEEK 40]
    (EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                       HOURS-WORKED-THIS-WEEK))
    (EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
    (LTQ 1 I)
    (FORALL Y (IMPLIES (AND (LTQ 1 Y)
                            (LTQ Y (SUBTRACT I 1)))
                       (EQ (SELECT HOURS-WORKED-WEEKLY-OUT
                                   (Y))
                           (SELECT HOURS-WORKED-WEEKLY-IN (Y]
  (IMPLIES
    (NOT (GT I 52))
    (AND
      [FORALL X (IMPLIES (AND (LTQ 1 X)
                              (LTQ X INPUT-MASTER-FILE.LENGTH))
                         (AND (EQ (SELECT NAME-IN.ARRAY (X))
                                  (SELECT (QUAL NAME.ARRAY
                                                TIME-CARD)
                                          (X)))
                              (LTQ 0 (SELECT CURRENT-WEEK-IN.ARRAY
                                             (X)))
                              (LTQ (SELECT CURRENT-WEEK-IN.ARRAY
                                           (X))
                                   51]
      (EQ INPUT-MASTER-FILE.LENGTH TIME-CARD-FILE.LENGTH)
      [FORALL
        X
        (IMPLIES
          (AND (LTQ 1 X)
               (LTQ X OUTPUT-MASTER-FILE.INDEX))
          (AND (EQ (SELECT NAME-IN.ARRAY (X))
                   (SELECT (QUAL NAME.ARRAY TIME-CARD)
                           (X))
                   (SELECT NAME-OUT.ARRAY (X))
                   (SELECT (QUAL NAME.ARRAY PAYCHECK)
                           (X)))
               [EQ (SELECT GROSS-PAY-TO-DATE-OUT.ARRAY (X))
                   (PLUS (SELECT GROSS-PAY-TO-DATE-IN.ARRAY
                                 (X))
                         (ROUND 999V99
                                (TIMES (SELECT
                                         WEEKLY-SALARY-IN.ARRAY
                                           (X))
                                       (DIVIDE (SELECT
                                  HOURS-WORKED-THIS-WEEK.ARRAY
```

BEST AVAILABLE COPY

```
                                                (X))
                                        40]
                    [EQ (SELECT HOURS-WORKED-TO-DATE-OUT.ARRAY (X))
                        (PLUS (SELECT HOURS-WORKED-TO-DATE-IN.ARRAY
                                      (X))
                              (SELECT HOURS-WORKED-THIS-WEEK.ARRAY
                                      (X]
                    [EQ (SELECT CURRENT-WEEK-OUT.ARRAY (X))
                        (PLUS 1 (SELECT CURRENT-WEEK-IN.ARRAY (X]
                    [FORALL Y (IMPLIES [AND (GTQ 1 Y)
                                            (LTQ Y 52)
                                            (NEQ Y (SELECT
                                              CURRENT-WEEK-OUT.ARRAY
                                                            (X]
                                       (EQ (SELECT (SELECT
                                       HOURS-WORKED-WEEKLY-OUT.ARRAY
                                                            (X))
                                                      (Y))
                                           (SELECT (SELECT
                                        HOURS-WORKED-WEEKLY-IN.ARRAY
                                                            (X))
                                                      (Y]
                    (EQ (SELECT (SELECT HOURS-WORKED-WEEKLY-OUT.ARRAY
                                        (X))
                                (SELECT CURRENT-WEEK-OUT.ARRAY (X)))
                        (SELECT HOURS-WORKED-THIS-WEEK.ARRAY (X]
(EQ (SUBTRACT INPUT-MASTER-FILE.LENGTH 1)
    (SUBTRACT TIME-CARD-FILE.INDEX 1)
    OUTPUT-MASTER-FILE.LENGTH PAYCHECK-FILE.LENGTH)
(EQ NAME-IN (QUAL NAME TIME-CARD)
    NAME-OUT
    (QUAL NAME PAYCHECK))
[EQ GROSS-PAY-TO-DATE-OUT
    (PLUS GROSS-PAY-TO-DATE-IN
          (ROUND 999V99 (TIMES WEEKLY-SALARY-IN
                               (DIVIDE HOURS-WORKED-THIS-WEEK
                                       40]
(EQ HOURS-WORKED-TO-DATE-OUT (PLUS HOURS-WORKED-TO-DATE-IN
                                   HOURS-WORKED-THIS-WEEK))
(EQ CURRENT-WEEK-OUT (PLUS 1 CURRENT-WEEK-IN))
(LTQ 1 I)
(FORALL Y (IMPLIES (AND (LTQ 1 Y)
                        (LTQ Y I))
                   (EQ (SELECT HOURS-WORKED-WEEKLY-IN
                               (Y))
                       (IF (EQ Y I)
                           (SELECT HOURS-WORKED-WEEKLY-IN
                                   (Y))
                           (SELECT HOURS-WORKED-WEEKLY-OUT
                                   (Y]
```

Table VIII-9

INFERENCE RULES FOR PROOF PRESENTED IN TABLE VIII-8

1. hyp.
2. $A_1 \wedge \ldots \wedge A_n \vdash A_i (1 \leq i \leq n)$, 1
3. inst. y, 2
4. hyp.
5. m.p., 3, 4
6. $B \vdash A \supset B$, 5
7. $A = A$
8. $B \vdash A \supset B$, 7
9. $A \supset x = y \wedge \neg A \supset x = z \equiv$

   $x = \underline{\text{if}}\ A\ \underline{\text{then}}\ y\ \underline{\text{else}}\ z$

   (definition of conditional expression), 6, 8
10. ded., 4, 9
11. hyp.
12. hyp.
13. $A, \neg A \vdash B$, 11, 12
14. ded., 12, 13
15. $A = A$
16. $B \vdash A \supset B$, 15
17. def. conditional expression, 16, 14
18. ded., 11, 17
19. $A \supset B.\ C \supset B \vdash A \vee C \supset B$, 10, 18
20. hyp.
21. $A \wedge B \vdash B$, 20
22. $x \leq y \supset x \leq y + 1 \vee x = y$ (x, y integers), 21
23. hyp.
24. $A \wedge B \vdash B$, 20
25. $A, B \vdash A \wedge B$, 23, 24
26. $B \vdash A \vee B$, 25
27. ded., 23, 26
28. hyp.
29. $B \vdash A \vee B$, 28

--BEST AVAILABLE COPY

30. ded., 28, 29

31. $A \vee C, A \supset B, C \supset B \vdash B$, 22, 27, 30

32. m.p., 19, 31

33. ded., 20, 32

34. gen. y, 33

35. $A \wedge B \vdash A$, 1

36. $A, B \vdash A \supset B$, 37, 36

37. $B \vdash A \supset B$, 38

38. ded., 1, 39

LEGEND

hyp.	hypothesis	
inst.	instantiation	$\forall x P(x) \vdash P(z)$ if Z is not free in P(x)
gen.	generalization	$P(x) \vdash \forall z P(z)$ if Z is not free in P(x)
m.p.	modus ponens	$A \supset B, A \vdash B$
ded.	deduction	$(A \vdash B) \vdash A \supset B$
sub.	substitution	according to free variable rules.

IX CONCLUSIONS

A. General

This work has shown the feasibility of the verification of COBOL programs in realistic application areas. The main result of the project has been to uncover some major remaining difficulties that must be resolved to make verification an effective tool.

Achievements of this project are:

(1) Decomposition of the verification process into stages, making the system easier to implement and to interact with.

(2) A very effective axiomatization of the COBOL data structures and control statements, that fits well into the structure of the verification system.

(3) A process that yields verification conditions that are simple to prove, although there are many verification conditions to prove, even for a simple program.

The major problems of COBOL verification as encountered in this project are:

(1) Verbosity of the programs, assertions, and verification conditions.

(2) The semantic complexity of the COBOL language.

These two problems have a trade-off in their solutions. A verification system that handles the semantic complexity directly makes the verification conditions less verbose, but a system that translates the complexity into simpler units makes the verification conditions more verbose.

The remaining research involved in COBOL verification should be aimed at making it possible to verify bigger, more complex COBOL programs, more easily and with more machine aid. This means doing basic research in techniques to structure the complexity of COBOL verification, while at the same time engineering the developing system to make things more convenient for the user. We believe that the following tasks would yield significant benefits in both of the above areas:

(1) Enlarging the subset of COBOL amenable to verification. This means dealing with the problem of character strings and their relation to numeric data.

(2) Engineering the parts of the system to make them easier to use.

(3) Research on deductive systems, including incremental simplification during posttransduction processing and verification condition generation.

(4) Work on techniques to help write COBOL programs that are easier to verify. These would include management techniques to restrict the kinds of programs written and possible minor syntactic changes to the language.

(5) Extension of the assertion language to make it easier to state abstract properties of COBOL programs.

(6) Exploration of the use of data abstraction techniques to enable the verification of large, structured COBOL programs.

(7) Development of an interactive COBOL environment closely coupled with the COBOL verification system.

B. A Note on a Programming Environment for COBOL

A COBOL verification system is not like a compiler, because a programmer cannot submit a program with assertions to the verifier and receive a verified program as output. Closely coupled interaction with the verification system is required at all stages of the verification process, and frequent changes to both programs and assertions are to be expected.

A program should be developed for verification. Operations required on these programs (e.g., formal testing, symbolic tracing, debugging) can be performed optimally under close interaction with the programmer. It has also been substantiated[25] that interactive programming improves program reliability and programmer productivity.

Finally, we feel that programmers will not write assertions for programs unless such a task is made easy for them by their programming environment. The need for tools designed for this purpose is great.

We feel that building these tools around an interactive interpreter for COBOL programs is the best way to proceed to the ultimate goal of making program verification usable. Even if the remaining research issues are resolved, it will take much time before the best environment for verification is achieved. We believe that enough is now understood about the nature of COBOL verification to enable an effective environment to be built.

X REFERENCES

1. L. Robinson and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Technical Report, Stanford Research Institute, Computer Science Group, Menlo Park, California (January 1975), submitted for publication.

2. L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena, "On Attaining Reliable Software for a Secure Operating System," Proceedings International Conference on Reliable Software, 21-23 April 1975, Los Angeles, California, pp. 267-284 (April 1975).

3. J. von Neumann and H. H. Goldstine, "Planning and Coding Problems for an Electronic Computer Instrument, Part II, Vol. 1-3" John von Neumann, Collected Works, Vol. 5, pp. 80-235, Pergamon Press, New York (1963).

4. R. W. Floyd, "Assigning Meanings to Programs," Proceedings American Mathematical Society Symposium in Applied Mathematics, Vol. 19, pp. 19-31 (1967).

5. P. Naur, "Proof of Algorithms by General Snapshots," BIT 6, pp.310-316 (1966).

6. B. Elspas, K. N. Levitt, and R. J. Waldinger, "Design of an Interactive System for Verification of Computer Programs," SRI Report, Project 1891, Stanford Research Institute, Menlo Park, California (July 1973).

7. S. Igarashi, R. London, and D. Luckham, "Automatic Verification of Programs I: A Logical Basis and Implementation," Memo AIM-200, Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, California (May 1973).

8. C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica 2, pp. 335-355 (1973).

9. S. Katz and Z. Manna, "Semantic Analysis of Programs," unpublished paper (1975).

10. E. W. Dijkstra, "Guarded Commands, Nondeterminacy, and a Calculus for the Derivation of Programs," Proceedings International Conference on Reliable Software, 21-23 April 1975, Los Angeles, California (1975).

11. R. L. Sites, "Clean Termination of Computer Programs," Ph.D. Dissertation, Stanford University, Stanford, California (June 1974).

12. American National Standard Programming Language COBOL. American National Standards Institute, New York (1974).

13. P. Wegner, "The Vienna Definition Language," Computing Surveys 4, 1, pp. 5-63 (March 1972).

14. CODASYL Data Base Task Group Report, Association for Computing Machinery, New York (April 1971).

15. W. Teitelman, INTERLISP Reference Manual,Xerox Palo Alto Research Center, Palo Alto, California (1974).

16. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, A Paged Time Sharing System for the PDP-10," Comm. ACM 15, 3, pp. 135-143 (March 1972).

17. B. Elspas, R. S. Boyer, R. E. Shostak, and J. M. Spitzen, "A Verification System for JOVIAL/J73 Programs," Draft Final Report, SRI Project 3756, Stanford Research Institute, Computer Science Group, Menlo Park, California (November 1975).

18. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," Comm. ACM 12, 10, pp. 576-581 (October 1969).

19. B. Liskov, "An Approach to Abstraction," MIT Project MAC, Computation Structures Group, Memo 88, Massachusetts Institute of Technology, Cambridge, Massachusetts (September 1973).

20. W. A. Wulf, "ALPHARD--Toward a Language to Support Structured Programming," Carnegie-Mellon University, Pittsburgh, Pennsylvania, unpublished paper (April 1974).

21. P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena, "A Provably Secure Operating System," Final Report, SRI Project 2581, Stanford Research Institute, Menlo Park, California (June 1975).

22. E. W. Dijkstra, "Notes on Structured Programming," Structured Programming, C. A. R. Hoare (ed.), Academic Press, New York (1972).

23. H. D. Mills, "How to Write Correct Programs and Know It," Proceedings International Conference on Reliable Software 21-23 April, Los Angeles, California, pp. 363-370 (April 1975).

24. E. Mendelson, Introduction to Mathematical Logic, Van Nostrand, Princeton, New Jersey (1964).

25. M. M. Gold, "Time-Sharing and Batch--An Experimental Comparison of their Values in a Problem-Solving Situation," Comm. ACM 12, 5 (May 1969).

26. A. P. Morse, A Theory of Sets, Academic Press, New York (1965).

GLOSSARY*

Abstraction - a technique for hiding particular parts of a phenomenon (e.g., a programs behavior) to make the other parts of that phenomenon easier to understand.

Assertion - a predicate in first-order logic concerning the values of variables in a program.

Assertion, inductive - an assertion placed within the program text to break up the program's flowchart into simple paths containing a fixed number of program statements.

Assertion, input - an assertion that constrains the values of a program's input data.

Assertion, output - an assertion that relates the values of a program's input data to the values of its output data.

Assertion language - first-order logic along with some predefined functions that express the semantics of domains that may be related to a programming language (e.g., integers, reals, strings, and arrays).

Checks, compile-time - any decidable restrictions on a program that can be placed on its source code, i.e., detectable from the syntax alone.

Checks, run-time - any restrictions on a program's execution that are not decidable until the program is executed with a particular set of input data.

Deductive system - a program that attempts (with or without user guidance) to generate a formal proof of a verification condition.

Language, real - a programming language that is used in some kind of software production.

Ordering, dynamic - ordering of the execution of a program's statements, which is dependent on the input data.

Ordering, lexical - ordering of statements in the source code of a program.

Path - a sequence of program statements that is executed for particular values of the input data; it can be infinite.

Path, simple - a path that executes a fixed, finite number of program statements for any values of the input data; a program's flowchart can be broken up into a set of simple paths.

Posttransduction processing - translation of a transduced form of a program into another internal form that is suitable for verification condition generation.

---

*Terms applying to COBOL alone are not defined here. Check Reference 12 for definition.

Program verification - the process of proving that the behavior of a program is consistent with an input assertion and an output assertion.

Proof checker - a simple program to check the output of a deductive system to see if the proof is logically sound.

Semantics - the rules that determine how any element of a language is interpreted, in terms of some model.

Simplification - application of algebraic and propositional rewrite rules to reduce the complexity of a formula.

Structured programming - a discipline for reducing the complexity of programs by using control primitives to guarantee nested flow-charts. (Note: this is not Dijkstra's definition of the term.)

Subset - a restriction of the syntax of a language that is also compatible with the language's semantics.

Syntax - the rules that determine for a given whether a character string is an element of a particular language.

Termination - a property of a program stating that the program finishes execution in finite time for particular input data.

Termination, clean - a property of a program that includes termination and absence of run-time errors.

Transduction - translation of a program into an internal form defined by a transduction grammar.

Transduction grammar - a set of rules defining both the syntax of a language and an algorithm for producing an internal form for a program in the language.

Verification condition - a logical formula, produced from a program in which assertions have been inserted, that is equivalent to the logical consistency of the program (according to some semantic model) and the assertions; often referred to in the plural (i.e., the verification conditions for a program).

Verification condition generator - a program that takes a program and assertions as input, and produces a verification condition.

APPENDIX

CODE FOR COBOL VERIFICATION SYSTEM

The Appendix contains two sections. The first section contains the INTERLISP code to manipulate transduction grammars, which was not developed during the current work but was used as part of the system. This code is not documented. The second section contains the INTERLISP code for the symbol table (ST), posttransduction processing (PTP), and verification condition generation (VC). The role of each function is briefly described, along with its affiliation with one of the three constituent modules above.

A. Code for Parsing and Transduction

```
(FILECREATED " 5-JAN-76 13:34:56" NEWPARSE.;3 16502
     previous date: " 5-JAN-76 11:30:47" NEWPARSE.;2)
  (LISPXPRINT (QUOTE NEWPARSECOMS)
              T T)
  (RPAQQ NEWPARSECOMS ((FNS * NEWPARSEFNS)
         (VARS TLIST OUTCOUNT FIRSTRHSCOL SECONDRHSCOL PRINTFLG)))
  (RPAQQ NEWPARSEFNS
         (ABSTRACT ADDFNS ADDON ADDSTATE COBOLTOKENFN COMPLETE
                   COMPLETEP COMPLIS COMPUTELOOK DELFNS DIF EARLY
                   ERASING-INDICES EXTRACT FIND-ERASING-RULES
                   FLUSHGRAMMAR FLUSHLEFT GETALT GETDOT GETDOTSYMBOL
                   GETLHS GETLOOK GETORIG GETORIGPTR GETPARSE GETRULE
                   GETTRAN JOVIALTOKENFN LEFTSET MAKEMATRIX MAKEPARSE
                   MAKEPARSE1 NTERMINALP OUTLINE POWER-SET PPC PREDICT
                   PREDICTP PRETTYGRAMMAR PRINTGRAMMAR PRINTGRAMMAR/R
                   PRINTSTATESET PURIFY PUTRULE PUTRULES PUTTRAN
                   PUTTRANS SAVEGRAM SCAN SCANP SETPARSE SORTRULES
                   TERMINALS TESTFINAL TRANSLATE))
(DEFINEQ
(ABSTRACT
  [LAMBDA (INPUT TOKENFN)
    (COND
      ((EQ (CAR (QUOTE NEWGRAMMARLOAD))
           (QUOTE NOBIND))
        (SETQ NEWGRAMMARLOAD T)
        (ATTACH (QUOTE ##ROOT)
                NONTERMS)
        (ADDON (QUOTE ##ROOT)
               (LIST (CADR NONTERMS)
                     (QUOTE RPAD)))
        (for NT in SPECIALNONTERMS do (DREMOVE NT NONTERMS)
                                      (REMPROP NT (QUOTE RULES))
                                      (REMPROP NT (QUOTE TRANS)))
        (SETPARSE)))
    (FLUSHLEFT)
    [FRPLACA (QUOTE INPUTSTRING)
             (APPEND INPUT (LIST (QUOTE RPAD]
    [for TOKEN in INPUT do (COND
                             ((NTERMINALP TOKEN)
                               (APPLY* TOKENFN TOKEN]
    (COND
      ((EARLY)
        (TRANSLATE (MAKEPARSE)))
      (T (QUOTE NOGO])
(ADDFNS
  [LAMBDA (X)
    (SETQ NEWPARSEFNS (APPEND NEWPARSEFNS X])
(ADDON
  [LAMBDA (LHS RHS)
    (COND
      ([NOT (MEMBER RHS (GETP LHS (QUOTE RULES]
        (PUTRULE LHS RHS)
        (PUTTRAN LHS (QUOTE T1])
(ADDSTATE
  [LAMBDA (LHS ALT DOT ORIG ORIGPTR LOOK PARSELIST)
    (PROG NIL
          (SETQ HASHITEM (PACK (LIST LHS (QUOTE #)
                                     ALT
                                     (QUOTE #)
                                     DOT
                                     (QUOTE #)
                                     ORIG)))
          (SETQ HASHVAL (GETHASH HASHITEM))
          (COND
            [HASHVAL (SETQ NEWLOOK (DIF LOOK HASHVAL))
                     (COND
                       (NEWLOOK (NCONC HASHVAL NEWLOOK))
                       (T (RETURN]
            (T (SETQ NEWLOOK LOOK)
               (PUTHASH HASHITEM LOOK)))
          (TCONC STATESET (NCONC (LIST (CONS LHS ALT)
                                       DOT
                                       (CONS ORIG ORIGPTR)
                                       NEWLOOK)
                                 PARSELIST])
(COBOLTOKENFN
  [LAMBDA (TOKEN)
    (COND
      ((NUMBERP TOKEN)
        (ADDON (QUOTE number)
               (LIST TOKEN)))
      ((STRINGP TOKEN)
        (ADDON (QUOTE string)
               (LIST TOKEN)))
      ((LITATOM TOKEN)
        (ADDON (QUOTE symbol)
               (LIST TOKEN)))
      ((EQ (CAR TOKEN)
           (QUOTE ASSERT))
        (ADDON (QUOTE assertion)
               (LIST TOKEN)))
      (T (ERROR "Unexpected token" TOKEN])
(COMPLETE
  [LAMBDA NIL
    (FOR STATE IN (GET (GETORIGPTR)
                       (GETLHS))
       DO (PROGN (FRPLACD (QUOTE TEMPSTORE)
                          (CDDDDR STATE))
                 (ADDPROP (QUOTE TEMPSTORE)
                          (ADD1 (CADR STATE))
                          (CAR STATEPTR)
                          T)
                 (ADDSTATE (CAAR STATE)
                           (CDAR STATE)
                           (ADD1 (CADR STATE))
                           (CAADDR STATE)
```

```
                                   (CDADDR STATE)
                                   (CADDDR STATE)
                                   (CDR (QUOTE TEMPSTORE])
(COMPLETEP
  [LAMBDA NIL
    (AND (NULL DOTSYMBOL)
         (FMEMB INPUTCHAR (GETLOOK])
(COMPLIS
  [LAMBDA (X Y)
    (COND
      ((NULL X)
        T)
      ((NULL Y)
        NIL)
      ((ALPHORDER (CAR X)
                  (CAR Y))
        (COND
          ((EQ (CAR X)
               (CAR Y))
            (COMPLIS (CDR X)
                     (CDR Y)))
          (T T)))
      (T NIL])
(COMPUTELOOK
  [LAMBDA NIL
    [SETQ TEMPSTORE (FNTH (GETRULE (GETLHS)
                                   (GETALT))
                          (PLUS 2 (GETDOT]
    (COND
      [TEMPSTORE (COND
                   ((GETP (CAR TEMPSTORE)
                          (QUOTE RULES))
                     (LEFTSET (CAR TEMPSTORE)))
                   (T (LIST (CAR TEMPSTORE]
      (T (GETLOOK])
(DELFNS
  [LAMBDA (X)
    (SETQ NEWPARSEFNS (FOR Z IN NEWPARSEFNS UNLESS (MEMB Z X)
                           COLLECT Z])
(DIF
  [LAMBDA (X Y)
    (FOR Z IN X UNLESS (FMEMB Z Y) COLLECT Z])
(EARLY
  [LAMBDA NIL
    (PROG NIL
          (SETQ STATESET (CONS))
          (CLRHASH)
          (ADDSTATE (CAR NONTERMS)
                    1 0 1 NIL (CONS (QUOTE RPAD)
                                    NIL))
          [FOR INPUTCHAR IN INPUTSTRING AS INPUTX FROM 1
             DO
              (PROGN [FOR STATEPTR ON (CAR STATESET)
                       DO (COND
                            ((PREDICTP)
                              (PREDICT))
                            ((COMPLETEP)
                              (COMPLETE]
                    [COND
                      [PRINTFLG
                        (COND
                          ((EQ PRINTFLG (QUOTE ALL))
                            (PRIN1 INPUTX)
                            (SPACES 2)
                            (COND
                              ((ILESSP (PRIN1 (LENGTH (CAR STATESET)))
                                       10)
                                (SPACES 1)))
                            (SPACES 2)
                            (PRINT INPUTCHAR]
                      (T (PRIN1 INPUTCHAR)
                         (COND
                           ((EQ INPUTCHAR (QUOTE RPAD))
                             (TERPRI))
                           (T (SPACES 3]
                    (SETQ OLDSTATESET STATESET)
                    (SETQ STATESET (CONS))
                    (CLRHASH)
                    [FOR STATEPTR ON (CAR OLDSTATESET)
                       DO (COND
                            ((SCANP)
                              (SCAN]
                    (COND
                      ((CAR STATESET)
                        (FRPLACD (QUOTE PREDLIST)
                                 NIL))
                      (T (PRINT (LIST (QUOTE STATESET)
                                      (ADD1 INPUTX)
                                      (QUOTE IS)
                                      (QUOTE EMPTY)))
                         (RETURN NIL]
          (TESTFINAL])
(ERASING-INDICES
  [LAMBDA (RHS)
    (for X in RHS as I from 1 bind (TEMP) when TEMP(FASSOC X
                                                     ERASING-RULES)
       collect <I ! TEMP>])
(EXTRACT
  [LAMBDA (NT RHS TRAN INDICES ANSWERS)
    (for TI
       in (QUOTE (T1 T2 T3 T4 T5 T6 T7 T8 T9 T10)) as RHST
       in RHS as N from 1
       bind ((SUB(LIST NIL))
             (DRULE(LIST NIL))
             (TI(QUOTE (T1 T2 T3 T4 T5 T6 T7 T8 T9 10)))
             TEMP
          do (COND
               [(SETQ TEMP (FASSOC N INDICES))
                 (TCONC SUB (CONS TI (CADDDR TEMP]
               (T (TCONC SUB (CONS TI (CAR TJ)))
                  (SETQ TJ (CDR TJ))
                  (TCONC DRULE RHST)))
          finally [TCONC ANSWERS (LIST (QUOTE PUTRULE)
                                       (KWOTE NT)
                                       (KWOTE (CAR DRULE]
                  (TCONC ANSWERS (LIST (QUOTE PUTTRAN)
                                       (KWOTE NT)
                                       (KWOTE (SUBLIS (CAR SUB)
                                                      TRAN T])
(FIND-ERASING-RULES
  [LAMBDA (NONTERMS)
    (for NT bind ((R <NIL>)) in NONTERMS
       do (for RHS in (GETP NT 'RULES) as TR in (GETP NT  TRANS)
             as I from 1 when RHS=NIL
             do (TCONC R <NT I TR>))
       finally (RETURN R:1])
(FLUSHGRAMMAR
  [LAMBDA NIL
    [COND
      ((NEQ (CAR (QUOTE NONTERMS))
            (QUOTE NOBIND))
        (PROGN (FOR NT IN NONTERMS DO (REMPROP NT (QUOTE RULES))
                                      (REMPROP NT (QUOTE TRANS)))
               (SETQ NONTERMS NIL]
    (RPAQQ NONTERMS NIL])
(FLUSHLEFT
  [LAMBDA NIL
    (FOR Z IN NONTERMS DO (REMPROP Z (QUOTE LEFTSET)))
    (FOR Z IN SPECIALNONTERMS DO (REMPROP Z (QUOTE LEFTSET])
(GETALT
  [LAMBDA NIL
    (CDAAR STATEPTR])
(GETDOT
  [LAMBDA NIL
    (CADAR STATEPTR])
(GETDOTSYMBOL
  [LAMBDA NIL
    [SETQ TEMPSTORE (FNTH (GETRULE (GETLHS)
                                   (GETALT))
                          (ADD1 (GETDOT]
    (COND
      (TEMPSTORE (CAR TEMPSTORE])
(GETLHS
  [LAMBDA NIL
    (CAAAR STATEPTR])
(GETLOOK
  [LAMBDA NIL
    (CADDDR (CAR STATEPTR])
(GETORIG
  [LAMBDA NIL
    (CAADDR (CAR STATEPTR])
(GETORIGPTR
  [LAMBDA NIL
    (CDADDR (CAR STATEPTR])
(GETPARSE
  [LAMBDA (STATE)
    (COND
      (STATE (CDDDDR STATE))
      (T (CDDDDR (CAR STATEPTR])
(GETRULE
  [LAMBDA (LHS ALT)
    (CAR (FNTH (GETP LHS (QUOTE RULES))
               ALT])
(GETTRAN
  [LAMBDA (LHS ALT)
    (CAR (NTH (GETP LHS (QUOTE TRANS))
              ALT])
(JOVIALTOKENFN
  [LAMBDA (TOKEN)
    (COND
      ((NUMBERP TOKEN)
        (ADDON (QUOTE number)
               (LIST TOKEN)))
      ((AND (LITATOM TOKEN)
            (EQ (NCHARS TOKEN)
                1))
        (ADDON (QUOTE letter)
               (LIST TOKEN)))
      ((LITATOM TOKEN)
        (ADDON (QUOTE symbol)
               (LIST TOKEN)))
      (T (ERROR "Unexpected token" TOKEN])
(LEFTSET
  [LAMBDA (X)
    (COND
      ((GETP X (QUOTE LEFTSET)))
      (T (PROG (SOFAR)
               (MAKEMATRIX X)
               [FOR NT IN SOFAR
                  DO (PROG (PTR)
                           (SETQ PTR (CONS))
                           (LCONC PTR (GETP NT (QUOTE LEFTSET)))
                           (FOR LSYM IN (CAR PTR)
                              DO (FOR Z IN (GETP LSYM (QUOTE LEFTSET))
                                    UNLESS (MEMB Z (CAR PTR))
                                    DO (TCONC PTR Z)))
                           (PUT NT (QUOTE LEFTSET)
                                (FOR Z IN (GETP NT (QUOTE LEFTSET))
                                   UNLESS (GETP Z (QUOTE RULES))
                                   COLLECT Z]
               (GETP X (QUOTE LEFTSET])
(MAKEMATRIX
  [LAMBDA (X)
    (PROG (LSYMLIST)
          (FOR RULEALT IN (GETP X (QUOTE RULES))
             UNLESS (MEMB (CAR RULEALT)
                          LSYMLIST)
             DO (SETQ LSYMLIST (CONS (CAR RULEALT)
                                     LSYMLIST)))
          (PUT X (QUOTE LEFTSET)
               LSYMLIST)
          (SETQ SOFAR (CONS X SOFAR))
          (FOR LSYM IN LSYMLIST WHEN (AND (GETP LSYM (QUOTE RULES))
                                          (NOT (MEMB LSYM SOFAR)))
             DO (MAKEMATRIX LSYM])
(MAKEPARSE
  [LAMBDA NIL
    (MAKEPARSE1 (CAR (CAR STATESET])
(MAKEPARSE1
  [LAMBDA (STATE)
    (CONS (CONS (CAAR STATE)
                (CDAR STATE))
          (FOR SYMBOL IN (GETRULE (CAAR STATE)
                                  (CDAR STATE))
             AS I FROM 1
             COLLECT (COND
                       ((NOT (GETP SYMBOL (QUOTE RULES)))
                         SYMBOL)
                       (T (MAKEPARSE1 (CAR (GET (CDDDDR STATE)
                                                I])
(NTERMINALP
  [LAMBDA (TOKEN)
    (NULL (GETP TOKEN (QUOTE TERMINAL])
(OUTLINE
  [LAMBDA (S)
    (COND
      [(EQ (CHCON1 S)
           13)
        (SETQ OUTCOUNT 0)
        (PRIN1 (PACKC (QUOTE (13 10]
      (T (SETQ OUTCOUNT (IPLUS OUTCOUNT (NCHARS S)))
         (PRIN1 S])
(POWER-SET
  [LAMBDA (S)
    (if S
        then (for X bind ((R <NIL>)) in (POWER-SET S::1)
                do (TCONC R X)
                   (TCONC R <S:1 ! X>)
                finally (RETURN R:1))
      else <NIL>])
(PPC
  [LAMBDA (P)
    (FOR X IN P WHEN (EQ (PROG1 (PRIN1 X)
                                (SPACES 1))
                         (QUOTE $.))
        DO (TERPRI])
 (PARSEIT
  (LAMBDA NIL
    (ADDPROP (QUOTE PREDLIST)
             DOTSYMBOL
             (CAR STATEPTR))
     (SETQ LOOKY (COMPUTELOOK))
     (FOR RULE IN RHSALTS AS I FROM 1 DO (ADDSTATE DOTSYMBOL I 0 INPUTX
                                                   (CDR (QUOTE PREDLIST))
                                                   LOOKY])
(PREDICTP
  (LAMBDA NIL
    (SETQ RHSALTS (GETP (SETQ DOTSYMBOL (GETDOTSYMBOL))
                        (QUOTE RULES])
(PRETTYGRAMMAR
  (LAMBDA (FIRSTRHSCOL SECONDRHSCOL)
    (COND
      ((NULL FIRSTRHSCOL)
        (COND
          ((EQ (CAR (QUOTE FIRSTRHSCOL))
               (QUOTE NOBIND))
            (SETQ FIRSTRHSCOL 20))
          (T (SETQ FIRSTRHSCOL (CAR (QUOTE FIRSTRHSCOL]
    (COND
      ((NULL SECONDRHSCOL)
        (COND
          ((EQ (CAR (QUOTE SECONDRHSCOL))
               (QUOTE NOBIND))
            (SETQ SECONDRHSCOL 40))
          (T (SETQ SECONDRHSCOL (CAR (QUOTE SECONDRHSCOL]
    [OUTLINE (PACKC (QUOTE (13]
    [FOR NT IN NONTERMS
       DO (OUTLINE NT)
          (OUTLINE (QUOTE % :=))
          (FOR [RHS RHST (RHSTX(GETP NT (QUOTE TRANS]
             IN (GETP NT (QUOTE RULES))
             DO (SETQ RHST (CAR RHSTX))
                (WHILE (ILESSP OUTCOUNT FIRSTRHSCOL)
                   DO (OUTLINE (QUOTE % )))
                (FOR E IN RHS DO (OUTLINE (QUOTE % ))
                                 (OUTLINE E))
                (COND
                  ((IGREATERP SECONDRHSCOL 0)
                    (WHILE (ILESSP OUTCOUNT SECONDRHSCOL)
                       DO (OUTLINE (QUOTE % )))
                    (PRIN1 RHST)))
                (OUTLINE (PACKC (QUOTE (13]
    (PACKC (QUOTE (0])
(PRINTGRAMMAR
  (LAMBDA (NONTERMS)
    (TERPRI)
    (TERPRI)
    (TERPRI)
    (TERPRI)
    (FOR NT IN NONTERMS
       DO (TERPRI)
          (PRINT NT)
          (FOR X IN (GETP NT (QUOTE RULES)) AS Y
             IN (GETP NT (QUOTE TRANS))
             DO (PRIN1 (QUOTE #))
                [COND
                  (X (SPACES 2)
                     (FOR TOKEN IN X DO (PRIN1 TOKEN)
                                        (SPACES 1]
                (TERPRI)
                (PRINTDEF (CLISPIFY Y T)
                          3)
                (TERPRI))
          (TERPRI)
          (PRINT (QUOTE __________________])
(PRINTGRAMMAR/R
  [LAMBDA (NONTERMS FILENAME TRANS)
    (COND
      (FILENAME (OUTFILE FILENAME)))
    (PRINT (QUOTE ))
    (PRIN1 (QUOTE .spacing% 1))
    (TERPRI)
    (PRINT (QUOTE .nofill))
    (PRINT (QUOTE .nojustify))
    (PRIN1 (QUOTE .tab% stops% 8,16,24,32,40))
    (TERPRI)
    (FOR NT IN NONTERMS
       DO (PRINT NT)
          [FOR X IN (GETP NT (QUOTE RULES)) AS Y
             IN (GETP NT (QUOTE TRANS))
             DO (PRIN1 (QUOTE =))
                [COND
                  (X (SPACES 2)
                     (FOR TOKEN IN X DO (PRIN1 TOKEN)
                                        (SPACES 1]
                (TERPRI)
                (COND
                  (TRANS (PRINTDEF (CLISPIFY Y T)
                                   3)
                         (TERPRI]
          (PRIN1 (QUOTE .blank% 1))
          (TERPRI)
          (PRINT (QUOTE -------------------------------------))
          (PRIN1 (QUOTE .blank% 1))
          (TERPRI))
    (COND
      (FILENAME (CLOSEF FILENAME])
(PRINTSTATESET
  [LAMBDA NIL
    (FOR STATEPTR ON (CAR STATESET)
       DO (PROGN (PRINT (LIST (GETLHS)
               (GETALT)
               (GETDOT)
               (GETORIG)
               (GETLOOK])
(PURIFY
 [LAMBDA (NONTERMS)
  (PROG ((ANSWERS (LIST NIL))
     (ERASING-RULES (FIND-ERASING-RULES NONTERMS)))
     [for NT in NONTERMS
       do (COND
           ([NEQ (FLENGTH (GETP NT (QUOTE RULES)))
                (FLENGTH (GETP NT (QUOTE TRANS]
            (HELP NT
' does not have the same number of transductions as right hand sides."))
           )
          (for RHS in (GETP NT (QUOTE RULES)) as TRAN
             in (GETP NT (QUOTE TRANS))
             to (for INDICES in (POWER-SET (ERASING-INDICES
                                                 RHS))
                  when INDICES
                  do (EXTRACT NT RHS TRAN INDICES ANSWERS]
     (for X in (CAR ANSWERS) do (EVAL X))
     (for ER in ERASING-RULES
       do (PUT (CAR ER)
               (QUOTE TRANS)
               (for X in (GETP (CAR ER)
                               (QUOTE RULES))
                  as Y in (GETP (CAR ER)
                                (QUOTE TRANS))
                  when X collect Y))
          (PUT (CAR ER)
               (QUOTE RULES)
               (for X in (GETP (CAR ER)
                               (QUOTE RULES))
                  when X collect X])
(PUTRULE
 [LAMBDA (LHS RHS)
  (ADDPROP LHS (QUOTE RULES)
        RHS)
  (COND
   ((OR (EQ (CAR (QUOTE NONTERMS))
            NOBIND)
        (NULL NONTERMS))
    (SETQ NONTERMS (LIST LHS)))
   (T (COND
        ((NOT (MEMB LHS NONTERMS))
         (NCONC1 NONTERMS LHS])
(PUTRULES
 [LAMBDA NUM
  (PROG (NT)
     (FOR I FROM 2 TO NUM WHEN (PUTRULE (ARG NUM 1)
                                         (ARG NUM I))
       DO (SETQ NT (ARG NUM 1)))
```

```
          (RETURN NT])
(PUTTRAN
  [LAMBDA (X TRAN)
    (ADDPROP X (QUOTE TRANS)
             TRAN])
(PUTTRANS
  [LAMBDA NUM
    (FOR I FROM 2 TO NUM DO (PUTTRAN (ARG NUM 1)
                                     (ARG NUM I])
(SAVEGRAM
  [LAMBDA (X SUPPRESS-SORT)
    [COND
      ((NOT SUPPRESS-SORT)
        (SORT (CDR NONTERMS]
    [SET (PACK (LIST X (QUOTE COMS)))
         (QUOTE ([COMS *(LIST (CONS (QUOTE IFPROP)
                                    (CONS (QUOTE (RULES TRANS))
                                          NONTERMS]
                 (VARS SPECIALNONTERMS NONTERMS SPECIALFNNAMES]
    (MAKEFILE X])
(SCAN
  [LAMBDA NIL
    (ADDSTATE (GETLHS)
              (GETALT)
              (ADD1 (GETDOT))
              (GETORIG)
              (GETORIGPTR)
              (GETLOOK)
              (GETPARSE])
(SCANP
  [LAMBDA NIL
    (EQ INPUTCHAR (GETDOTSYMBOL])
(SETPARSE
  [LAMBDA NIL
    (PUT (QUOTE RPAD)
         (QUOTE TERMINAL)
         (QUOTE RPAD))
    (for NT in NONTERMS
       do (for RULE in (GETP NT (QUOTE RULES))
             do (for TOKEN in RULE
                   do (COND
                        ((AND (NOT (MEMB TOKEN NONTERMS))
                              (NOT (MEMB TOKEN SPECIALNONTERMS)))
                          (PUT TOKEN (QUOTE TERMINAL)
                               TOKEN])
(SORTRULES
  [LAMBDA (NONTERMS)
    (FOR NT IN NONTERMS BIND PAIRS
       DO (SETQ PAIRS (FOR X IN (GETP NT (QUOTE RULES)) AS Y
                         IN (GETP NT (QUOTE TRANS))
                         COLLECT (CONS X Y)))
          [SORT PAIRS (FUNCTION (LAMBDA (A B)
                    (COMPLIS (CAR A)
                                        (CAR B]
                    (PUT NT (QUOTE RULES)
                         (FOR X IN PAIRS COLLECT (CAR X)))
                    (PUT NT (QUOTE TRANS)
                         (FOR Y IN PAIRS COLLECT (CDR Y])
(TERMINALS
  [LAMBDA NIL
    (PROG (ALLRHS)
          [FOR NT IN NONTERMS DO (FOR RULE IN (GETP NT (QUOTE RULES))
                                      DO (SETQ ALLRHS (APPEND RULE ALLRHS]
          (FOR NT IN NONTERMS DO (DREMOVE NT ALLRHS))
          (SORT ALLRHS)
          (RETURN (FOR (TOKEN LASTTOKEN) IN ALLRHS
                       WHEN (NEQ TOKEN LASTTOKEN) COLLECT (SETQ LASTTOKEN
                                                                 TOKEN)
                                                          TOKEN])

(TESTFINAL
  [LAMBDA NIL
    (PROGN (SETQ STATEPTR (CAR STATESET))
           (COND
             ((AND (EQ (GETLHS)
                       (CAR NONTERMS))
                   (EQ (GETALT)
                       1)
                   (EQ (GETDOT)
                       2))
               (QUOTE SUCCESS!])
(TRANSLATE
  [LAMBDA (P)
    (COND
      ((NLISTP P)
        P)
      ((EQ (CAR P)
           (QUOTE ASSERT))
        P)
      (T (APPLY (LIST (QUOTE LAMBDA)
                      TLIST
                      (GETTRAN (CAAR P)
                               (CDAR P)))
                (FOR PX IN (CDR P) COLLECT (TRANSLATE PX])
)
  (RPAQQ TLIST (T1 T2 T3 T4 T5 T6 T7 T8 T9 T10))
  (RPAQQ OUTCOUNT 0)
  (RPAQQ FIRSTRHSCOL 12)
  (RPAQQ SECONDRHSCOL -1)
  (RPAQQ PRINTFLG ALL)
(DECLARE: DONTCOPY
  (FILEMAP (NIL (891 16329 (ABSTRACT 903 . 1566) (ADDFNS 1570 . 1639)
(ADDON 1643 . 1777) (ADDSTATE 1781 . 2393) (COBOLTOKENFN 2397 . 2796)
(COMPLETE 2800 . 3206) (COMPLETEP 3210 . 3293) (COMPLIS 3297 . 3532)
(COMPUTELOOK 3536 . 3821) (DELFNS 3825 . 3927) (DIF 3931 . 4000) (EARLY
4004 . 5209) (ERASING-INDICES 5213 . 5367) (EXTRACT 5371 . 6057) (
FIND-ERASING-RULES 6061 . 6304) (FLUSHGRAMMAR 6308 . 6559) (FLUSHLEFT
6503 . 6706) (GETALT 6710 . 6755) (GETDOT 6759 . 6804) (GETDOTSYMBOL
6808 . 6966) (GETLHS 6970 . 7015) (GETLOOK 7019 . 7071) (GETORIG 7075
 . 7127) (GETORIGPTR 7131 . 7186) (GETPARSE 7190 . 7293) (GETRULE 7297
 . 7382) (GETTRAN 7386 . 7469) (JOVIALTOKENFN 7473 . 7814) (LEFTSET
7818 . 8385) (MAKEMATRIX 8389 . 8827) (MAKEPARSE 8831 . 8894) (
MAKEPARSE1 8898 . 9196) (NTERMINALP 9200 . 9271) (OUTLINE 9275 . 9460)
(POWER-SET 9464 . 9641) (PPC 9645 . 9762) (PREDICT 9766 . 10013) (
PREDICTP 10017 . 10117) (PRETTYGRAMMAR 10121 . 11104) (PRINTGRAMMAR
11108 . 11582) (PRINTGRAMMAR/R 11586 . 12393) (PRINTSTATESET 12397
 . 12582) (PURIFY 12586 . 13525) (PUTRULE 13529 . 13799) (PUTRULES
13803 . 13982) (PUTTRAN 13986 . 14058) (PUTTRANS 14062 . 14159) (
SAVEGRAM 14163 . 14472) (SCAN 14476 . 14638) (SCANP 14642 . 14698)
(SETPARSE 14702 . 15040) (SORTRULES 15044 . 15450) (TERMINALS 15454
 . 15841) (TESTFINAL 15845 . 16068) (TRANSLATE 16072 . 16326)))))
STOP
```

B. Documentation and Code for Verification Condition Generation and Posttransduction Processing

1. Documentation

#ADDCORRESPONDING$ [SENTENCE; PAIR;]
remarks: PTP - Processes an ADD CORRESPONDING statement.
called by: SENTENCESCAN

CORRESPAIRS [X,Y; XSONS, YSONS, XSON, YSON;]
remarks: PTP - Returns list of dotted pairs of corresponding elementary items of X and Y.
called by: #ADDCORRESPONDING$, CORRESPAIRS, #MOVECORRESPONDING$, #SUBTRACTCORRESPONDING$

SONS [QUALNAME; X;]
remarks: ST - Returns the list of sons for a qualified data item.
called by: CORRESPAIRS, ELEMITEMSOF

COMPLETELIST [QUALNAME; X;]
remarks: ST - Returns the completely qualified name of a data item.
called by: SONS, PICTURE*, LEVEL*, OCCURS*, VALUE*

QUAL [QUALNAME;;]
remarks: ST - Appends (QUAL) to a data name if it is unqualified.
called by: COMPLETELIST, AMBIGUOUS

AMBIGUOUS [NAME; X;]
remarks: ST - returns T if a name is an ambiguous data reference, NIL otherwise.
called by: COMPLETELIST

QUALIFIEROK [QUALLIST, PREDLIST; Z;]
remarks: ST - returns T if a qualified name (QUALLIST) is not in conflict with a predecessor list (PREDLIST).
called by: AMBIGUOUS, QUALIFIEROK, COMPLETELIST, MULTIPLE

#ASSERT [SENTENCE;;]
remarks: PTP - Processes an ASSERT statement.
called by: SENTENCESCAN

ASSERT1 [SENTENCE, PTR; X;]
remarks: PTP - Processes an ASSERT statement.
called by: #ASSERT, ASSERT1

MAXSIZE* [QUALNAME;;]
remarks: ST - Returns the maximum size of a qualified data item.
called by: ASSERT1, #SET$, #SETROUNDED$

MAXSIZE [PIC; X,Y,L,R;]
remarks: ST - Returns the maximum size of a PICTURE specification.
called by: MAXSIZE*

PICTURE* [QUALNAME;;]
remarks: ST - Returns the PICTURE specification for a qualified data name.
called by: MAXSIZE*

#IF [SENT; ARG1, ARG2;]
remarks: PTP - Processes an IF statement.
called by: SENTENCESCAN

SENTENCESCAN [TEXT; SENT;]
remarks: PTP - Scans a program to do post-transduction processing.
called by: #IF, SENTENCESCAN, #PERFORM, PREPROCESS, VC

#OPENIN [L;;]
remarks: PTP - Processes an OPEN INPUT statement.
called by: SENTENCESCAN, #OPENOUT

#OPENOUT [L;;]
remarks: PTP - Processes an OPEN OUTPUT statement.
called by: SENTENCESCAN

#READ [L; NAME, ITEM;]
remarks: PTP - Processes a READ statement.
called by: SENTENCESCAN

ELEMITEMSOF [X;;]
remarks: PTP - Returns the elementary items of a group data item.
called by: #READ, ELEMITEMSOF, #WRITE

RECORDLIST [NAME;;]
remarks: ST - Returns the list of records associated with a given file name.
called by: #READ, RECORDNAME

#WRITE [L; INDEXNAME, LENGTHNAME, ITEM;]
remarks: PTP - Processes a WRITE statement.
called by: SENTENCESCAN

FILE [NAME;;]
remarks: ST - Returns the file associated with a record.
called by: #WRITE

#PERFORM [SENT; ASSRT, INDEX, FIRSTVALUE, STEP, TERMINATION; FLATSECTEXT]
remarks: PTP - Processes a PERFORM statement.
called by: SENTENCESCAN

FLATTENPARAS [ABPROG; PARA;]
remarks: PTP - Eliminates paragraph structure from a COBOL program producing a list of sentences.
called by: #PERFORM, PREPROCESS, VC

GATHERPARAS [L1, L2, TEXT; X, Y;]
remarks: PTP - Gathers text in TEXT between the labels L1 and L2.
called by: #PERFORM

#SET$ [SENTENCE;;]
remarks: PTP - Processes a SET$ (assignment) statement.
called by: SENTENCESCAN

#SETROUNDED$ [SENTENCE;;]
remarks: PTP - Processes a SETROUNDED$ (rounded assignment) statement.
called by: SENTENCESCAN

```
#MOVECORRESPONDING$ [SENTENCE; PAIR; ]
     remarks:  PTP - Processes a MOVE CORRESPONDING statement.
     called by:  SENTENCESCAN

#SUBTRACTCORRESPONDING$ [SENTENCE; PAIR;]
     remarks:  PTP - Processes a SUBTRACT CORRESPONDING statement.
     called by:  SENTENCESCAN

CHANGELABEL [SENTENCE;; SECLIST]
     remarks:  PTP - Replaces labels in sentences by their fully-qualified
               (i.e., <paragraph name, section name>) versions.
     called by:  CHANGELABEL, LABELMAKER

GETNEWLABEL [LABEL;; SECLIST, SECTION]
     remarks:  PTP - Returns the fully-qualified version of a given label.
     called by:  CHANGELABEL

ERR [X;;]
     remarks:  PTP - Error routine. Prints argument.
     called by:  GETNEWLABEL

LASTPARA [SEC; SECTION; ABPROG]
     remarks:  PTP - Returns the fully-qualified label of the last
               paragraph in a given section.
     called by:  CHANGELABEL

COBOLVCG [PATHLIST; PATH;]
     remarks:  VC - Actual verification condition generator operating on
               the output of the path analyzer.
     called by:  VC

VCG1 [PATH, FORM;;]
     remarks:  VC - Recursive auxiliary function used by COBOLVCG.
     called by:  COBOLVCG, VCG1

CONVERT [PIC; X,Y,Z1,Z,$$TEM1, $$TEM2,Z2;]
     remarks:  ST - Canonicalizes a PICTURE specification.
     called by:  INSERTDATA, INSERT77ITEM

DEBUGPRINT [WHAT, WHERE, WHEN;;FLAT]
     remarks:  PTP - Auxiliary debugging function.
     called by:  PREPROCESS

PPR is not defined.

ELEMENTARYP [DATADESCRIPTION;;]
     remarks:  ST - Returns T if a variable name represents an
               elementary data item, NIL otherwise.
     called by:  GETRECORD

ERASETABLE [; X, Y: SYMBOLTABLE, TYPES, VALUES]
     remarks:  ST - Initializes symbol table.
     called by:
```

FETCHLABELASSERTIONS [ABPROG; PARA; LABELASSERTLIST]
    remarks: ST - Constructs A-list (LABELASSERTLIST) whose entries are of form <paraname, assertion> of all labelled assertions.
    called by: VC

ISASSERT [X;;]
    remarks: PTP - Predicate testing for list of form (ASSERT.....)
    called by: FETCHLABELASSERTIONS, PATHANAL

FILENAME [RECORD;;]
    remarks: PTP - Returns file name corresponding to a given record
    called by:

FLATTENSECTIONS [ABPROG; SECTION;]
    remarks: PTP - Flattens abstract program into a list of paragraphs.
    called by: PREPROCESS, VC

GETRECORD [GARBAGE, LEVELLIST, NAMELIST; CURRENTLEVEL, CURRENTRECORD, DUMMY, $$TEM1, $$TEM2;]
    remarks: ST - Constructs a tree-structured data declaration out of a flat list.
    called by: GETRECORD, GETRECORD*

INSERTDATA [NAME, PREDECESSORS, PICTURE, LEVEL, SONLIST, OCCURS, VALUE]
    remarks: ST - Inserts a data item into the symbol table.
    called by: GETRECORD

NNULLATOM [A;;]
    remarks: ST - Returns T if the argument is not a null atom, NIL otherwise.
    called by: INSERTDATA

PICTUREOK [PIC; X; PICTURECHARS]
    remarks: ST - Returns T if PIC is a permissible PICTURE specification NIL otherwise.
    called by: INSERTDATA, INSERT77ITEM

MULTIPLE [NAME, QUALS; X;]
    remarks: ST - Returns T if NAME and QUALS are conflicting qualified data names.
    called by: INSERTDATA, INSERT77ITEM

INSERTVALUE [FULLNAME, VALUEEXP;; VALUES]
    remarks: ST - Inserts a value designation into the symbol value.
    called by: INSERTDATA, INSERT77ITEM

INSERTSYMBOL [NAME;;SYMBOLTABLE]
    remarks: ST - Inserts a variable into the symbol table.
    called by: INSERTDATA, INSERT77ITEM, INSERTFILE, INSERTPARAGRAPH, INSERTRECORD, INSERTSECTION

GETSONS [TREE; X;]
    remarks: ST - Returns list of names of sons of a group data item.
    called by: GETRECORD

GETRECORD* [RECORDLIST; X, 77DECS, DATADECS, Y;]
remarks: ST - Takes a list containing possibly many data declarations and turns it into a declaration bill.
called by:

INSERT77ITEM [NAME, PICTURE, OCCURS, VALUE;;]
remarks: ST - Inserts a 77-item into the symbol table.
called by: GETRECORD*

INSERTFILE [NAME, RECORDLIST;;]
remarks: ST - Inserts a file name, and its corresponding record list, into the symbol table.
called by:

INSERTPARAGRAPH [NAME, SECTION;;]
remarks: ST - Inserts a paragraph name into the symbol table.
called by:

INSERTRECORD [NAME, FILE;;]
remarks: ST - Inserts a record name into the symbol table.
called by:

INSERTSECTION [NAME;;]
remarks: ST - Inserts a section name into the symbol table.
called by:

LABELMAKER [ABPROG; SECLIST, PARALIST, SECTION, PARAGRAPH, SECTION, PARAGRAPH, PARANAME, SECNAME;]
remarks: PTP - Replaces all label references (and labels) in program by their fully-qualified [paraname, sectionname] versions
called by: PREPROCESS, VC

LEVEL* [QUALNAME;;]
remarks: ST - Returns the level number of a qualified data item.
called by:

OCCURS* [QUALNAME;;]
remarks: ST - Returns the number of occurrences of a qualified data item.
called by:

PATHANAL [SENTLIST, ACCUM, SENT; X; PATHLIST, TEMP, LABELASSERTLIST]
remarks: VC - Constructs from list of sentences a list (PATHLIST) of paths in the program, each path beginning and ending with an assertion.
called by: PATHANAL, VC

PREPROCESS [ABPROG, FLAG; FLATSECTEXT, FLATPARATEXT, TEXT;]
remarks: PTP - Transforms abstract program, readying it for path analysis (Now obsolete, superceded by VC).
called by:

RECORDNAME [FILE;;]
    remarks: ST - Returns the record name corresponding to a file.
    called by:

SECTIONLIST [NAME;;]
    remarks: ST - Returns the list of sections for a given paragraph name.
    called by:

SECTIONP [NAME;;]
    remarks: ST - Returns T if the argument is a section name, NIL otherwise.
    called by:

VALUE* [QUALNAME;;]
    remarks: ST - Returns the VALUE expression for a qualified data item.
    called by:

VC [ABPROG; PATHLIST, FLATSECTEXT, LABELASSERTLIST; SCANOUT, PATHANALOUT, VCOUT]
    remarks: VC & PTP - Takes raw abstract program as input; performs preprocessing, path generation, and verification condition generation. Returns list (VCOUT) of verification conditions.
    called by:

ZAP [;; ZZZ, MIKE/A]
    remarks: PTP - Internal debugging routine.
    called by:

2. Code

```
 (FILECREATED "10-JAN-76 17:34:56" PREPSYMBOL.;4 28553
      changes to:  LABELMAKER SENTENCESCAN
      previous date: " 9-JAN-76 20:51:25" PREPSYMBOL.;3)
   (LISPXPRINT (QUOTE PREPSYMBOLCOMS)
               T T)
   [RPAQQ PREPSYMBOLCOMS ((FNS * PREPSYMBOLFNS)
           (DECLARE: DONTEVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVARS
                     (ADDVARS (NLAMA)
                              (NLAML DEBUGPRINT]
   (RPAQQ PREPSYMBOLFNS
          (#ADDCORRESPONDING$ #ASSERT #IF #MOVECORRESPONDING$ #OPENIN
                              #OPENOUT #PERFORM #READ #SET$ #SETROUNDED$
                              #SUBTRACTCORRESPONDING$ #WRITE AMBIGUOUS
                              ASSERT1 CHANGELABEL COBOLVCG COMPLETELIST
                              CONVERT CORRESPAIRS DEBUGPRINT ELEMENTARYP
                              ELEMITEMSOF ERASETABLE ERR
                              FETCHLABELASSERTIONS FILE FILENAME
                              FLATTENPARAS FLATTENSECTIONS GATHERPARAS
                              GETNEWLABEL GETRECORD GETRECORD* GETSONS
                              INSERT77ITEM INSERTDATA INSERTFILE
                              INSERTPARAGRAPH INSERTRECORD INSERTSECTION
                              INSERTSYMBOL INSERTVALUE ISASSERT
                              LABELMAKER LASTPARA LEVEL* MAXSIZE
                              MAXSIZE* MULTIPLE NULLATOM OCCURS*
                              PATHANAL PICTURE* PICTUREOK PREPROCESS
                              QUAL QUALIFIEROK RECORDLIST RECORDNAME
                              SECTIONLIST SECTIONP SENTENCESCAN SONS
                              VALUE* VC VCG1 ZAP))
(DEFINEQ
(#ADDCORRESPONDING$
  [LAMBDA (SENTENCE)
    (FOR PAIR IN (CORRESPAIRS (CADR SENTENCE)
                              (CADDR SENTENCE))
        COLLECT (LIST (COND
                        ((CADDDR SENTENCE)
                          (QUOTE SETROUNDED$))
                        (T (QUOTE SET$)))
                      (CDR PAIR)
                      (LIST (QUOTE PLUS)
                            (CDR PAIR)
                            (CAR PAIR))
                      (CAR (CDDDDR SENTENCE])
(#ASSERT
  [LAMBDA (SENTENCE)
    (ASSERT1 SENTENCE)
    (LIST SENTENCE])
(#IF
  [LAMBDA (SENT)
    (PROG (ARG1 ARG2)
          [SETQ ARG1 (SENTENCESCAN (LIST (CADDR SENT]
          [SETQ ARG2 (SENTENCESCAN (LIST (CADDDR SENT]
          [COND
            ((CDR ARG1)
```

BEST AVAILABLE COPY

```
              (SETQ ARG1 (LIST (QUOTE BLOCK)
                               ARG1)))
            (T (SETQ ARG1 (CAR ARG1]
          [COND
            ((CDR ARG2)
              (SETQ ARG2 (LIST (QUOTE BLOCK)
                               ARG2)))
            (T (SETQ ARG2 (CAR ARG2]
          (RETURN (LIST (LIST (QUOTE IF)
                              (CADR SENT)
                              ARG1 ARG2])
(#MOVECORRESPONDING$
  [LAMBDA (SENTENCE)
    (FOR PAIR IN (CORRESPAIRS (CADR SENTENCE)
                              (CADDR SENTENCE))
       COLLECT (LIST (QUOTE SET$)
                     (CDR PAIR)
                     (CAR PAIR)
                     NIL])
(#OPENIN
  [LAMBDA (L)
    (LIST (LIST (QUOTE SET$)
                (MKATOM (CONCAT (CADR L)
                                (QUOTE .INDEX)))
                0])
(#OPENOUT
  [LAMBDA (L)
    (#OPENIN L])
(#PERFORM
  [LAMBDA (SENT)
    (PROG ((ASSRT (CADDR (CDDR SENT)))
           INDEX FIRSTVALUE STEP TERMINATION)
          (COND
            [(EQ (CADR SENT)
                 (QUOTE TIMES))
              (SETQ INDEX (GENSYM))
              (RETURN
                (LIST
                  (CONS
                    (QUOTE BLOCK)
                    (SENTENCESCAN
                      (APPEND [LIST (LIST (QUOTE SET$)
                                          INDEX 1)
                                    ASSRT
                                    (LIST (QUOTE IF)
                                          (LIST (QUOTE GT)
                                                INDEX
                                                (CADDDR SENT))
                                          (QUOTE (ENDPERFORM))
                                          (QUOTE (NEXT]
                              (FLATTENPARAS (GATHERPARAS
                                                (CADR (CADDR SENT))
                                                (CADDR (CADDR SENT))
```

```
                                              FLATSECTEXT))
                              (LIST (LIST (QUOTE SET$)
                                          INDEX
                                          (LIST (QUOTE PLUS)
                                                INDEX 1))
                                    (LIST (QUOTE LOOPASSERT)
                                          ASSRT]
            [(EQ (CADR SENT)
                 (QUOTE VARYING))
              (SETQ INDEX (CAR (CADDDR SENT)))
              (SETQ FIRSTVALUE (CADR (CADDDR SENT)))
              (SETQ STEP (CADDR (CADDDR SENT)))
              (SETQ TERMINATION (CADDDR (CADDDR SENT)))
              (RETURN
                (LIST
                  (CONS
                    (QUOTE BLOCK)
                    (SENTENCESCAN
                      (APPEND [LIST (LIST (QUOTE SET$)
                                          INDEX FIRSTVALUE)
                                    ASSRT
                                    (LIST (QUOTE IF)
                                          TERMINATION
                                          (QUOTE (ENDPERFORM))
                                          (QUOTE (NEXT]
                              (FLATTENPARAS (GATHERPARAS
                                              (CADR (CADDR SENT))
                                              (CADDR (CADDR SENT))
                                              FLATSECTEXT))
                              (LIST (LIST (QUOTE SET$)
                                          INDEX
                                          (LIST (QUOTE PLUS)
                                                INDEX STEP))
                                    (LIST (QUOTE LOOPASSERT)
                                          (LIST (QUOTE ASSERT)
                                                ASSRT]
            [(EQUAL (CADR SENT)
                    (QUOTE (ONCE$)))
              (RETURN
                (LIST (CONS (QUOTE BLOCK)
                            (SENTENCESCAN
                              (FLATTENPARAS (GATHERPARAS
                                              (CADR (CADDR SENT))
                                              (CADDR (CADDR SENT))
                                              FLATSECTEXT]
            (T (QUOTE (STRANGE SYNTAX IN BLOCK SENTENCE]))
(#READ
   (LAMBDA (L)
   (PROG [(NAME (MKATOM (CONCAT (CADR L)
                                 (QUOTE .INDEX]
          (NCONC
            (LIST (LIST (QUOTE SET$)
                        NAME
```

```
                       (LIST (QUOTE PLUS)
                             NAME 1))
                 (LIST (QUOTE IF)
                       [LIST (QUOTE GT)
                             NAME
                             (MKATOM (CONCAT (CADR L)
                                             (QUOTE .LENGTH]
                       (CADDDR L)
                       (QUOTE (NEXT]
           (FOR ITEM IN [ELEMITEMSOF (CAR (RECORDLIST (CADR L]
              COLLECT
               (LIST (QUOTE SET$)
                     ITEM
                     (LIST (QUOTE SELECT)
                           (CONS (QUOTE QUAL)
                                 (CONS (MKATOM (CONCAT (CADR ITEM)
                                                       (QUOTE .ARRAY)))
                                       (CDDR ITEM)))
                           (LIST NAME))
                     NIL])
(#SET$
  [LAMBDA (SENTENCE)
    (LIST (COND
            [(CADDDR SENTENCE)
              (LIST (QUOTE IF)
                    (LIST (QUOTE GT)
                          (LIST (QUOTE ABS)
                                (LIST (QUOTE TRUNCATE)
                                      (CADR SENTENCE)
                                      (CADDR SENTENCE)))
                          (MAXSIZE* (CADR SENTENCE)))
                    (CADDDR SENTENCE)
                    (LIST (QUOTE ASSIGN)
                          (CADR SENTENCE)
                          (LIST (QUOTE TRUNCATE)
                                (CADR SENTENCE)
                                (CADDR SENTENCE]
             (T (LIST (QUOTE ASSIGN)
                      (CADR SENTENCE)
                      (LIST (QUOTE TRUNCATE)
                            (CADR SENTENCE)
                            (CADDR SENTENCE])
(#SETROUNDED$
  [LAMBDA (SENTENCE)
    (LIST (COND
            [(CADDDR SENTENCE)
              (LIST (QUOTE IF)
                    (LIST (QUOTE GT)
                          (LIST (QUOTE ABS)
                                (LIST (QUOTE ROUND)
                                      (CADR SENTENCE)
                                      (CADDR SENTENCE)))
                          (MAXSIZE* (CADR SENTENCE)))
                    (CADDDR SENTENCE)
                    (LIST (QUOTE ASSIGN)
                          (CADR SENTENCE)
                          (LIST (QUOTE ROUND)
                                (CADR SENTENCE)
                                (CADDR SENTENCE]
            (T (LIST (QUOTE ASSIGN)
                     (CADR SENTENCE)
                     (LIST (QUOTE ROUND)
                           (CADR SENTENCE)
                           (CADDR SENTENCE])
(#SUBTRACTCORRESPONDING$
  [LAMBDA (SENTENCE)
    (FOR PAIR IN (CORRESPAIRS (CADR SENTENCE)
                              (CADDR SENTENCE))
       COLLECT (LIST (COND
                       ((CADDDR SENTENCE)
                         (QUOTE SETROUNDED$))
                       (T (QUOTE SET$)))
                     (CDR PAIR)
                     (LIST (QUOTE SUBTRACT)
                           (CDR PAIR)
                           (CAR PAIR))
                     (CAR (CDDDDR SENTENCE])
(#WRITE
  [LAMBDA (L)
    (PROG [[INDEXNAME (MKATOM (CONCAT (FILE (CADR L))
                                      (QUOTE .INDEX]
           (LENGTHNAME (MKATOM (CONCAT (FILE (CADR L))
                                       (QUOTE .LENGTH]
          (NCONC
            (LIST (LIST (QUOTE SET$)
                        INDEXNAME
                        (LIST (QUOTE PLUS)
                              INDEXNAME 1))
                  (LIST (QUOTE SET$)
                        LENGTHNAME
                        (LIST (QUOTE PLUS)
                              LENGTHNAME 1)))
            (FOR ITEM IN (ELEMITEMSOF (CADR L))
               COLLECT
                (LIST (QUOTE SET$)
                      (LIST (QUOTE SELECT)
                            (CONS (QUOTE QUAL)
                                  (CONS (MKATOM (CONCAT (CADR ITEM)
                                                        (QUOTE .ARRAY)))
                                        (CDDR ITEM)))
                            (LIST INDEXNAME))
                      ITEM NIL])
(AMBIGUOUS
  [LAMBDA (NAME)
          (* accepts a name, either qualified or unqualified,
          and returns t if it is an ambiguous data reference)
```

```
      (SETQ NAME (QUAL NAME))
      (GREATERP (FOR X IN (GETP (CADR NAME)
                                (QUOTE ELEM$))
                   COUNT (QUALIFIEROK (CDDR NAME)
                                      (CAR X)))
                 1])
  (ASSERT1
    [LAMBDA (SENTENCE PTR)
      (COND
        ((NULL SENTENCE))
        ((ATOM SENTENCE))
        [(EQ (CAR SENTENCE)
             (QUOTE MAXSIZE))
          (RPLACA PTR (MAXSIZE* (CADR SENTENCE]
        (T (FOR X ON (CDR SENTENCE) DO (ASSERT1 (CAR X)
                                                 X])
  (CHANGELABEL
    [LAMBDA (SENTENCE)
      (COND
        ((NULL SENTENCE))
        ((ATOM SENTENCE))
        (T (SELECTQ (CAR SENTENCE)
                    (READ (CHANGELABEL (CADDDR SENTENCE)))
                    (IF (CHANGELABEL (CADDR SENTENCE))
                        (CHANGELABEL (CADDDR SENTENCE)))
                    [GO (RPLACA (CDR SENTENCE)
                                (GETNEWLABEL (CADR SENTENCE]
                    (SET$ (CHANGELABEL (CADDDR SENTENCE)))
                    (SETROUNDED$ (CHANGELABEL (CADDDR SENTENCE)))
                    [DO$ (RPLACA (CDR SENTENCE)
                                 (GETNEWLABEL (CADR SENTENCE)))
                         (RPLACA (CDDR SENTENCE)
                                 (COND
                                   ((MEMB (CADDR SENTENCE)
                                          SECLIST)
                                     (LASTPARA (CADDR SENTENCE)))
                                   (T (GETNEWLABEL (CADDR SENTENCE]
                    (PERFORM (CHANGELABEL (CADDR SENTENCE)))
                    NIL])
  (COBOLVCG
    [LAMBDA (PATHLIST)
      (CONS (QUOTE AND)
            (FOR PATH IN PATHLIST COLLECT (VCG1 (CDR PATH)
                                                (CADAR PATH])
  (COMPLETELIST
    [LAMBDA (QUALNAME)
      (SETQ QUALNAME (QUAL QUALNAME))
      (COND
        ((AMBIGUOUS QUALNAME)
          (HELP QUALNAME "ambiguous reference'))
        (T (FOR X IN (GETP (CADR QUALNAME)
                           (QUOTE ELEM$))
              UNTIL (QUALIFIEROK (CDDR QUALNAME)
                               (CAR X))
            FINALLY (COND
                      [X (RETURN (CONS (CADR QUALNAME)
                                       (CAR X]
                      (T (HELP QUALNAME "is not in symbol table"])
(CONVERT
  [LAMBDA (PIC)
    (IF (NULL PIC)
        THEN NIL
      ELSE
       (PROG (X (Y (LIST NIL))
                Z1)
             (SETQ X (UNPACK PIC))
             (WHILE (LISTP X)
                DO (SELECTQ (CAR X)
                            ((9 V P S X)
                              (NCONC1 Y (CAR X))
                              (SETQ X (CDR X)))
                            [%( (SETQ X (CDR X))
                                (SETQ Z1
                                  (PACK (FOR Z IN OLD X
                                           UNTIL (EQ Z (QUOTE %)))
                                           COLLECT Z)))
                                (SETQ X (CDR X))
                                (IF (NOT (NUMBERP Z1))
                                    THEN (HELP Z1
                                            "bad iterative picture"))
                                (PROG [(Z2 (CAR (LAST Y]
                                       (RPTQ (SUB1 Z1)
                                             (NCONC1 Y Z2]
                             (HELP PIC "bad picture specification")))
              (RETURN (PACK (CDR Y])
(CORRESPAIRS
  [LAMBDA (X Y)
    (PROG (XSONS YSONS)
           (SETQ XSONS (SONS X))
           (SETQ YSONS (SONS Y))
           (COND
             [(AND XSONS YSONS)
               (RETURN (FOR XSON IN XSONS
                          JOIN (FOR YSON IN YSONS
                                  WHEN (EQ (CADR XSON)
                                           (CADR YSON))
                                  JOIN (CORRESPAIRS XSON YSON]
             ((OR XSONS YSONS)
               NIL)
             (T (RETURN (LIST (CONS X Y])
(DEBUGPRINT
  [NLAMBDA (WHAT WHERE WHEN)
    (PROG NIL
           (COND
             ((MEMB WHEN FLAG)
               (TERPRI)
```

```
                  (PRINT (QUOTE -----------------------------------))
                  (TERPRI)
                  (PRINT WHERE)
                  (TERPRI)
                  (TERPRI)
                  (PPR (EVAL WHAT)))
                (T NIL])
(ELEMENTARYP
  [LAMBDA (DATADESCRIPTION)                       (* tells whether a given
                                                  data description is of
                                                  an elementary item)
    (CADDR DATADESCRIPTION])
(ELEMITEMSOF
  [LAMBDA (X)
    (COND
      ((FOR SON IN (SONS X) JOIN (ELEMITEMSOF SON)))
      (T (LIST X])
(ERASETABLE
  [LAMBDA NIL
          (* initializes the symboltable by clearing the
          property lists of all names in the symboltable, then
          clears the variable symboltable.
          must be done when performing two parses in a row)
    (FOR X IN SYMBOLTABLE DO (FOR Y IN TYPES DO (REMPROP X Y)))
    (SETQ SYMBOLTABLE)
    (SETQ VALUES])
(ERR
  [LAMBDA (X)
    (PRINT X)
    (RETFROM])
(FETCHLABELASSERTIONS
  [LAMBDA (ABPROG)
    (FOR PARA IN ABPROG WHEN (ISASSERT (CADDR PARA))
       DO (SETQ LABELASSERTLIST (CONS (CONS (CADR PARA)
                                            (CADDR PARA))
                                      LABELASSERTLIST])
(FILE
  [LAMBDA (NAME)                                  (* returns the file
                                                  corresponding to a given
                                                  record name)
    (GETP NAME (QUOTE RECORD$])
(FILENAME
  [LAMBDA (RECORD)
    (QUOTE FILENAME])
(FLATTENPARAS
  [LAMBDA (ABPROG)
    (FOR PARA IN ABPROG JOIN (APPEND (CDDR PARA])
(FLATTENSECTIONS
  [LAMBDA (ABPROG)
    (FOR SECTION IN (CDR ABPROG) JOIN (APPEND (CDDR SECTION])
(GATHERPARAS
  [LAMBDA (L1 L2 TEXT)
    (FOR X ON TEXT WHEN (EQUAL (CADAR X)
                                         L1)
      DO (RETURN (FOR Y ON (REVERSE X) WHEN (EQUAL (CADAR Y)
                                                   L2)
                     DO (RETURN (REVERSE Y])
(GETNEWLABEL
  [LAMBDA (LABEL)
    (COND
      [(ATOM LABEL)
        (COND
          ((MEMB LABEL SECLIST)
            (LIST (GETP LABEL (QUOTE FIRSTPARA))
                  LABEL))
          ((MEMB (CADR SECTION)
                 (GETP LABEL (QUOTE SECS)))
            (LIST LABEL (CADR SECTION)))
          [(EQUAL (LENGTH (GETP LABEL (QUOTE SECS)))
                  1)
            (LIST LABEL (CAR (GETP LABEL (QUOTE SECS]
          (T (ERR (QUOTE (NON-UNIQUE LABEL REFERENCE]
      (T LABEL])
(GETRECORD
  [LAMBDA (GARBAGE LEVELLIST NAMELIST)
          (* this function takes a flat list of record
          descriptions and turns them into a tree -- in the
          COBOL sense. it iterates through the list...)
    (IF (NLISTP GARBAGE)
        THEN (HELP NIL " bad call of GETRECORD"))
    (PROG (CURRENTLEVEL CURRENTRECORD DUMMY)
          (IF (NLISTP (CAR GARBAGE))
              THEN (HELP (CAR GARBAGE)
                         "bad record description"))
          (SETQ CURRENTLEVEL (CAAR GARBAGE))
          (IF LEVELLIST
              THEN (IF (ILESSP CURRENTLEVEL (CAR LEVELLIST))
                       THEN (HELP (CAAR GARBAGE)
                                  " bad record
structure")))
          (CONS (EACHTIME (SETQ CURRENTRECORD (CAR GARBAGE))
                 WHILE (AND GARBAGE (NOT (ILESSP (CAR CURRENTRECORD)
                                                 CURRENTLEVEL)))
                 COLLECT (IF (AND (EQP (CAR CURRENTRECORD)
                                       CURRENTLEVEL)
                                  (ELEMENTARYP CURRENTRECORD))
                             THEN
          (* here is an elementary item at the current level
          -- IT IS placed in the symboltable and the pointer
          is moved up)
                                  (INSERTDATA (CADR CURRENTRECORD)
                                              NAMELIST
                                              (CADDR CURRENTRECORD)
                                              CURRENTLEVEL NIL
                                              (CAR (NTH CURRENTRECORD
                                                        4))
```

```
                                                       (CAR (NTH CURRENTRECORD
                                                                 5)))
                                            (SETQ GARBAGE (CDR GARBAGE))
                                            CURRENTRECORD
                                     ELSEIF (EQP (CAR CURRENTRECORD)
                                                 CURRENTLEVEL)
                                       THEN
                   (* here is a group item at the current level.
                   it is put in the symboltable, and the function is
                   called recursively to handle the elementary items
                   that will follow. after returning, the global
                   variable is reset and the subtree is gathered into
                   the fold)
                                            (SETQ DUMMY
                                              (GETRECORD (SETQ GARBAGE
                                                               (CDR GARBAGE))
                                                         (CONS CURRENTLEVEL
                                                               LEVELLIST)
                                                         (CONS (CADR
                                                              CURRENTRECORD)
                                                               NAMELIST)))
                                            (INSERTDATA (CADR CURRENTRECORD)
                                                        NAMELIST NIL
                                                        CURRENTLEVEL
                                                        (GETSONS (CAR DUMMY))
                                                        (CAR (NTH CURRENTRECORD
                                                                  4))
                                                        (CAR (NTH CURRENTRECORD
                                                                  5)))
                                            (SETQ GARBAGE (CDR DUMMY))
                                            (LIST CURRENTRECORD (CAR DUMMY))
                                     ELSE (HELP NIL " bad record structure')))
                          GARBAGE])
(GETRECORD*
  [LAMBDA (RECORDLIST)                                 (* separates
                                                       77declarations from
                                                       tree-structured
                                                       declarations and calls
                                                       getrecord on the
                                                       tree-structured
                                                       declarations)
    (FOR (X (77DECS(LIST NIL))
            (DATADECS(LIST NIL)))
       IN RECORDLIST DO (IF (EQP (CAR X)
                                 77)
                            THEN (NCONC1 77DECS X)
                          ELSE (NCONC1 DATADECS X))
       FINALLY [FOR Y IN (SETQ 77DECS (CDR 77DECS))
                  DO (INSERT77ITEM (CADR Y)
                                   (CADDR Y)
                                   (CADDDR Y)
                                   (CAR (NTH Y 5]
               (RETURN (APPEND (CAR (GETRECORD (CDR DATADECS)))
                                        77DECS])
(GETSONS
  [LAMBDA (TREE)
    (FOR X IN TREE COLLECT (IF (NLISTP (CAR X))
                                 THEN (CADR X)
                               ELSE (CADAR X])
(INSERT77ITEM
  [LAMBDA (NAME PICTURE OCCURS VALUE)
          (* inserts a 77-item together with its picture.
          checks for non-unique references, bad picture,
          modifies property list and symboltable.)
    (IF (MULTIPLE NAME)
        THEN (HELP NAME "multiply defined 77-item")
      ELSEIF (OR (NULL (SETQ PICTURE (CONVERT PICTURE)))
                 (NOT (PICTUREOK PICTURE)))
        THEN (HELP NAME "bad picture specification")
      ELSEIF (NOT (ATOM OCCURS))
        THEN (HELP OCCURS "bad occurs expression")
      ELSEIF (LISTP VALUE)
        THEN (HELP VALUE "bad value statement")
      ELSE (ADDPROP NAME (QUOTE ELEM$)
                    (CONS NIL (LIST PICTURE 77 NIL OCCURS VALUE)))
           (INSERTVALUE NAME VALUE)
           (INSERTSYMBOL NAME])
(INSERTDATA
  [LAMBDA (NAME PREDECESSORS PICTURE LEVEL SONLIST OCCURS VALUE)
          (* inserts an entry for a group or elementary data
          item. checks for malformed predecessor list, bad
          picture, and non-unique data references.)
    (IF (NNULLATOM PREDECESSORS)
        THEN (HELP (CONS NAME (LIST PREDECESSORS))
                   "incorrect predecessor list")
      ELSEIF [NOT (PICTUREOK (SETQ PICTURE (CONVERT PICTURE]
        THEN (HELP (CONS NAME (LIST PREDECESSORS))
                   "incorrect picture specification")
      ELSEIF (MULTIPLE NAME PREDECESSORS)
        THEN (HELP (CONS NAME (LIST PREDECESSORS))
                   "multiply defined elementary item")
      ELSEIF (NOT (NUMBERP LEVEL))
        THEN (HELP LEVEL "improper level specification")
      ELSEIF (NNULLATOM SONLIST)
        THEN (HELP SONLIST "not a list of sons")
      ELSEIF (NOT (ATOM OCCURS))
        THEN (HELP OCCURS "bad occurs expression")
      ELSEIF (LISTP VALUE)
        THEN (HELP VALUE " bad value statement")
      ELSE (ADDPROP NAME (QUOTE ELEM$)
                    (CONS PREDECESSORS (LIST PICTURE LEVEL SONLIST
                                             OCCURS VALUE)))
           (INSERTVALUE (APPEND (LIST (QUOTE QUAL)
                                      NAME)
                                PREDECESSORS)
                        VALUE)
```

```
                (INSERTSYMBOL NAME])
(INSERTFILE
  [LAMBDA (NAME RECORDLIST)
                (* inserts list of records pertaining to a given
                file. checks for malformed recordlist and multiple
                references.)
    (IF (NLISTP RECORDLIST)
        THEN (HELP NIL 'IMPROPER RECORDLIST")
      ELSEIF (GETP NAME (QUOTE FILE$))
        THEN (HELP NAME "DUPLICATE RECORD DECLARATION )
      ELSE (PUT NAME (QUOTE FILE$)
                RECORDLIST)
                (INSERTSYMBOL NAME])
(INSERTPARAGRAPH
  [LAMBDA (NAME SECTION)
                (* inserts a section for a given paragraph.
                two paragraphs of the same name must be in different
                sections.)
    (IF (FMEMB SECTION (GETP NAME (QUOTE PARAGRAPH$)))
        THEN (HELP NAME "APPEARS TWICE IN A SECTION")
      ELSE (ADDPROP NAME (QUOTE PARAGRAPH$)
                    SECTION)
                (INSERTSYMBOL NAME])
(INSERTRECORD
  [LAMBDA (NAME FILE)                           (* inserts a recordname
                                                into the symboltable,
                                                with its corresponding
                                                file)
    (IF (GETP NAME (QUOTE RECORD$))
        THEN (HELP NAME "MULTIPLY DEFINED RECORD")
      ELSE (PUT NAME (QUOTE RECORD$)
                FILE)
                (INSERTSYMBOL NAME])
(INSERTSECTION
  [LAMBDA (NAME)                                (* inserts a section
                                                name into the
                                                symboltable)
    (IF (GETP NAME (QUOTE SECTION$))
        THEN (HELP NAME "MULTIPLY DEFINED SECTION')
      ELSE (PUT NAME (QUOTE SECTION$)
                (QUOTE T))
                (INSERTSYMBOL NAME])
(INSERTSYMBOL
  [LAMBDA (NAME)                                (* adds a symbol to the
                                                global variable that
                                                represents the symbol
                                                table)
    [SETQ SYMBOLTABLE (SORT (CONS NAME (DREMOVE NAME SYMBOLTABLE]
    NAME])
(INSERTVALUE
  [LAMBDA (FULLNAME VALUEEXP)
    (IF VALUEEXP
        THEN (SETQ VALUES (CONS (CONS FULLNAME VALUEEXP)
                                       VALUES])
(ISASSERT
  [LAMBDA (X)
    (AND (LISTP X)
         (EQ (CAR X)
             (QUOTE ASSERT])
(LABELMAKER
  [LAMBDA (ABPROG)
    (PROG (SECLIST PARALIST)
          [FOR SECTION IN (CDR ABPROG)
             DO (PUT (CADR SECTION)
                     (QUOTE FIRSTPARA)
                     (CADADR (CDR SECTION)))
                (SETQ SECLIST (CONS (CADR SECTION)
                                    SECLIST))
                (FOR PARAGRAPH IN (CDDR SECTION)
                   DO (SETQ PARALIST (CONS (CADR PARAGRAPH)
                                           PARALIST))
                      (ADDPROP (CADR PARAGRAPH)
                               (QUOTE SECS)
                               (CADR SECTION]
          [FOR SECTION IN (CDR ABPROG)
             DO (FOR PARAGRAPH IN (CDDR SECTION)
                   DO (RPLACA (CDR PARAGRAPH)
                              (LIST (CADR PARAGRAPH)
                                    (CADR SECTION)))
                      (FOR SENTENCE IN (CDDR PARAGRAPH)
                         DO (CHANGELABEL SENTENCE]
          (FOR PARANAME IN PARALIST DO (REMPROP PARANAME (QUOTE SECS)))
          (FOR SECNAME IN SECLIST DO (REMPROP SECNAME (QUOTE FIRSTPARA])
(LASTPARA
  [LAMBDA (SEC)
    (FOR SECTION IN (CDR ABPROG) WHEN (EQ (CADR SECTION)
                                          SEC)
       DO (RETURN (LIST (CADAR (LAST SECTION))
                        SEC])
(LEVEL*
  [LAMBDA (QUALNAME)
    (SETQ QUALNAME (COMPLETELIST QUALNAME))
    (CADDR (SASSOC (CDR QUALNAME)
                   (GETP (CAR QUALNAME)
                         (QUOTE ELEM$])
(MAXSIZE
  [LAMBDA (PIC)
          (* returns the largest value that will fit in the
          picture corresponding to a given qualified variable)
    (FOR (X YNIL
            LO
            RO)
       IN (UNPACK PIC) DO (SELECTQ X
                                   (S)
                                   (V (SETQQ Y T))
                                   [9 (IF Y
                                                       THEN (SETQ R (SUB1 R))
                                                     ELSE (SETQ L (ADD1 L]
                                                [P (IF Y
                                                       THEN (SETQ L (SUB1 L))
                                                            (SETQ R (SUB1 R))
                                                     ELSE (SETQ L (ADD1 L))
                                                          (SETQ R (ADD1 R]
                                                (HELP PIC "incorrect picture"))
          FINALLY (RETURN (DIFFERENCE (EXPT 10 L)
                                      (EXPT 10 R])
(MAXSIZE*
  [LAMBDA (QUALNAME)                                   (* gives the maximum
                                                       size of the value of a
                                                       qualified variable)
    (MAXSIZE (PICTURE* QUALNAME])
(MULTIPLE
  [LAMBDA (NAME QUALS)
    (OR [FOR X IN (GETP NAME (QUOTE ELEM$))
           THEREIS (OR (QUALIFIEROK (CAR X)
                                    QUALS)
                       (QUALIFIEROK QUALS (CAR X]
        (FMEMB NAME QUALS])
(NNULLATOM
  [LAMBDA (A)
    (AND A (NLISTP A])
(OCCURS*
  [LAMBDA (QUALNAME)
    (SETQ QUALNAME (COMPLETELIST QUALNAME))
    (CAR (NTH (SASSOC (CDR QUALNAME)
                      (GETP (CAR QUALNAME)
                            (QUOTE ELEM$)))
              5])
(PATHANAL
  [LAMBDA (SENTLIST ACCUM SENT)
    (COND
      ((NULL SENTLIST)
        (LIST ACCUM))
      (T
        (SETQ SENT (CAR SENTLIST))
        (SELECTQ
          (CAR SENT)
          (NEXT (PATHANAL (CDR SENTLIST)
                          ACCUM))
          (ASSERT (SETQ ACCUM (CONS SENT ACCUM))
                  (SETQ PATHLIST (CONS ACCUM PATHLIST))
                  (PATHANAL (CDR SENTLIST)
                            (LIST SENT)))
          [STOP (SETQ ACCUM (CONS (CADR SENT)
                                  ACCUM))
                (SETQ PATHLIST (CONS ACCUM PATHLIST))
                (SETQ SENTLIST (CDR SENTLIST))
                (COND
                  (SENTLIST (COND
                                    [(ISASSERT (CAR SENTLIST))
                                      (PATHANAL (CDR SENTLIST)
                                                (LIST (CAR SENTLIST]
                                    (T NIL)))
                        (T (QUOTE (END]
                [GO (SETQ TEMP (SASSOC (CADR SENT)
                                       LABELASSERTLIST))
                    [COND
                      ((NULL TEMP)
                        (HELP (QUOTE (GOTO TARGET HAS NO ASSERTION]
                    (SETQ ACCUM (CONS (CDR TEMP)
                                      ACCUM))
                    (SETQ PATHLIST (CONS ACCUM PATHLIST))
                    (SETQ SENTLIST (CDR SENTLIST))
                    (COND
                      (SENTLIST (COND
                                  [(ISASSERT (CAR SENTLIST))
                                    (PATHANAL (CDR SENTLIST)
                                              (LIST (CAR SENTLIST]
                                  (T NIL)))
                      (T (QUOTE (END]
                [IF (UNION (PATHANAL (CONS (CADDR SENT)
                                           (CDR SENTLIST))
                                     (CONS (LIST (QUOTE IF)
                                                 (CADR SENT))
                                           ACCUM))
                           (PATHANAL (CONS (CADDDR SENT)
                                           (CDR SENTLIST))
                                     (CONS (LIST (QUOTE IF)
                                                 (LIST (QUOTE NOT)
                                                       (CADR SENT)))
                                           ACCUM]
                [BLOCK (UNION (FOR X IN (PATHANAL (CDR SENT)
                                                  ACCUM)
                                 JOIN (COND
                                        ((EQ X (QUOTE END))
                                          (COND
                                            [(ISASSERT (CADR SENTLIST))
                                              (PATHANAL (CDDR SENTLIST)
                                                        (LIST (CADR SENTLIST]
                                            (T NIL)))
                                        (T (PATHANAL (CDR SENTLIST)
                                                     X]
                (LOOPASSERT (SETQ ACCUM (CONS (CADR SENT)
                                              ACCUM))
                            (SETQ PATHLIST (CONS ACCUM PATHLIST))
                            (LIST (QUOTE END)))
                (ENDPERFORM (LIST ACCUM))
                (PATHANAL (CDR SENTLIST)
                          (CONS (CAR SENTLIST)
                                ACCUM])
           (PICTURE*
             (LAMBDA (QUALNAME)
        (SETQ QUALNAME (COMPLETELIST QUALNAME))
        (CADR (SASSOC (CDR QUALNAME)
                      (GETP (CAR QUALNAME)
                            (QUOTE ELEM$])
(PICTUREOK
  [LAMBDA (PIC)
    (OR (NULL PIC)
        (AND (ATOM PIC)
             (FOR X IN (UNPACK PIC) ALWAYS (FMEMB X PICTURECHARS])
(PREPROCESS
  [LAMBDA (ABPROG FLAG)
    (PROG (FLATSECTEXT FLATPARATEXT (TEXT (COPY ABPROG)))
          (DEBUGPRINT TEXT INPUT-PROGRAM 1)
          (LABELMAKER TEXT)
          (DEBUGPRINT TEXT AFTER-LABELMAKER 2)
          (SETQ FLATSECTEXT (FLATTENSECTIONS TEXT))
          (DEBUGPRINT FLATSECTEXT AFTER-FLATTENSECTION 3)
          (SETQ FLATPARATEXT (FLATTENPARAS FLATSECTEXT))
          (DEBUGPRINT FLATPARATEXT AFTER-FLATTENPARAS 4)
          (SETQ FLATPARATEXT (SENTENCESCAN FLATPARATEXT))
          (DEBUGPRINT FLATPARATEXT AFTER-SENTENCESCAN-IN-PREPROCESS 5])
(QUAL
  [LAMBDA (QUALNAME)
    (COND
      ((NLISTP QUALNAME)
        (LIST (QUOTE QUAL)
              QUALNAME))
      ((NEQ (CAR QUALNAME)
            (QUOTE QUAL))
        (HELP QUALNAME "improper qualifier"))
      (T QUALNAME])
(QUALIFIEROK
  [LAMBDA (QUALLIST PREDLIST)
    (COND
      ((NLISTP QUALLIST)
        T)
      ((NLISTP PREDLIST)
        NIL)
      (T (PROG (Z)
               (SETQ Z (FMEMB (CAR QUALLIST)
                              PREDLIST))
               (RETURN (COND
                         ((NLISTP Z)
                           NIL)
                         (T (QUALIFIEROK (CDR QUALLIST)
                                         (CDR Z])
(RECORDLIST
  [LAMBDA (NAME)
    (GETP NAME (QUOTE FILE$])
(RECORDNAME
  [LAMBDA (FILE)
    (CAR (RECORDLIST FILE])
(SECTIONLIST
  [LAMBDA (NAME)
    (GETP NAME (QUOTE PARAGRAPH$])
(SECTIONP
  [LAMBDA (NAME)
    (GETP NAME (QUOTE SECTION$])
(SENTENCESCAN
  [LAMBDA (TEXT)
    (PROG NIL
          (FOR SENT IN TEXT
             JOIN
              (APPEND (SELECTQ
                         (CAR SENT)
                         (OPENINPUT$ (SENTENCESCAN (#OPENIN SENT)))
                         (OPENOUTPUT$ (SENTENCESCAN (#OPENOUT SENT)))
                         (READ (SENTENCESCAN (#READ SENT)))
                         (WRITE (SENTENCESCAN (#WRITE SENT)))
                         (IF (#IF SENT))
                         (PERFORM (#PERFORM SENT))
                         (SET$ (#SET$ SENT))
                         (SETROUNDED$ (#SETROUNDED$ SENT))
                         (MOVECORRESPONDING$ (SENTENCESCAN (
#MOVECORRESPONDING$ SENT)))
                         (ADDCORRESPONDING$ (SENTENCESCAN (
#ADDCORRESPONDING$ SENT)))
                         (SUBTRACTCORRESPONDING$ (SENTENCESCAN
                                                     (
#SUBTRACTCORRESPONDING$ SENT)))
                         [LOOPASSERT (LIST (CONS (QUOTE LOOPASSERT)
                                                 (SENTENCESCAN
                                                   (CDR SENT]
                         (ASSERT (#ASSERT SENT))
                         (DISPLAY NIL)
                         (ACCEPT NIL)
                         (LIST SENT])
(SONS
  [LAMBDA (QUALNAME)
    (SETQ QUALNAME (COMPLETELIST QUALNAME))
    (FOR X IN [CADDDR (SASSOC (CDR QUALNAME)
                              (GETP (CAR QUALNAME)
                                    (QUOTE ELEM$]
       COLLECT (APPEND (LIST (QUOTE QUAL)
                             X)
                       QUALNAME])
(VALUE*
  [LAMBDA (QUALNAME)
    (SETQ QUALNAME (COMPLETELIST QUALNAME))
    (CAR (NTH (SASSOC (CDR QUALNAME)
                      (GETP (CAR QUALNAME)
                            (QUOTE ELEM$)))
              6])
(VC
  [LAMBDA (ABPROG)
    (PROG (PATHLIST FLATSECTEXT LABELASSERTLIST)
     (LABELMAKER ABPROG)
     (SETQ FLATSECTEXT (FLATTENSECTIONS ABPROG))
     (FETCHLABELASSERTIONS FLATSECTEXT)
     (SETQ SCANOUT (SENTENCESCAN (FLATTENPARAS FLATSECTEXT)))
     (PATHANAL (CDR SCANOUT)
          (LIST (CAR SCANOUT)))
     (SETQ PATHANALOUT PATHLIST)
     (SETQ VCOUT (COBOLVCG PATHLIST))
     (RETURN VCOUT])
(VCG1
 [LAMBDA (PATH FORM)
  (COND
   ((NULL PATH)
    FORM)
   (T (SELECTQ (CAAR PATH)
         (ASSERT (LIST (QUOTE IMPLIES)
                (CADAR PATH)
                FORM))
         (IF (VCG1 (CDR PATH)
              (LIST (QUOTE IMPLIES)
                 (CADAR PATH)
                 FORM)))
         (ASSIGN (VCG1 (CDR PATH)
                (SUBST (COND
                    ((AND (LISTP (CADAR PATH))
                       (EQ (CAADAR PATH)
                         (QUOTE SELECT)))
                     (LIST (QUOTE CHANGE)
                        (CADR (CADAR PATH))
                        (CADDR (CADAR PATH))
                        (CADDAR PATH)))
                    (T (CADDAR PATH)))
                   (CADAR PATH)
                   FORM)))
         (VCG1 (CDR PATH)
            FORM])
(ZAP
 [LAMBDA NIL
  (PROG NIL
     (SETQ ZZZ (COPY MIKE/A))
     (RETURN])
)
[DECLARE: DONTEVAL@LOAD DOEVAL@COMPILE DONTCOPY COMPILERVARS
 (ADDTOVAR NLAMA)
 (ADDTOVAR NLAML DEBUGPRINT)
]
(DECLARE: DONTCOPY
 (FILEMAP (NIL (1274 28412 (#ADDCORRESPONDING$ 1286 . 1628) (#ASSERT
1632 . 1708) (#IF 1712 . 2202) (#MOVECORRESPONDING$ 2206 . 2410) (
#OPENIN 2414 . 2524) (#OPENOUT 2528 . 2570) (#PERFORM 2574 . 4408)
(#READ 4412 . 5041) (#SET$ 5045 . 5587) (#SETROUNDED$ 5591 . 6131)
(#SUBTRACTCORRESPONDING$ 6135 . 6486) (#WRITE 6490 . 7138) (AMBIGUOUS
7142 . 7466) (ASSIGN1 7470 . 7725) (CHANGELABEL 7729 . 8454) (COBOLVCG
8458 . 8589) (COMPLETELIST 8593 . 9005) (CONVERT 9009 . 9667) (
CORRESPAIRS 9671 . 10052) (DEBUGPRINT 10056 . 10346) (ELEMENTARYP
10350 . 10608) (ELEMITEMSOF 10612 . 10724) (ERASETABLE 10728 . 11098)
(ERR 11102 . 11152) (FETCHLABELASSERTIONS 11156 . 11361) (FILE 11365
 . 11609) (FILENAME 11613 . 11665) (FLATTENPARAS 11669 . 11753) (
FLATTENSECTIONS 11757 . 11856) (GATHERPARAS 11860 . 12063) (GETNEWLABEL
12067 . 12481) (GETRECORD 12485 . 14789) (GETRECORD* 14793 . 15671)
(GETSONS 15675 . 15800) (INSERT77ITEM 15804 . 16538) (INSERTDATA 16542
 . 17749) (INSERTFILE 17753 . 18183) (INSERTPARAGRAPH 18137 . 18568)
(INSERTRECORD 18572 . 19014) (INSERTSECTION 19018 . 19392) (INSERTSYMBOL
19396 . 19761) (INSERTVALUE 19765 . 19891) (ISASSERT 19895 . 19977)
(LABELMAKER 19981 . 20811) (LASTPARA 20815 . 20970) (LEVEL* 20974
 . 21130) (MAXSIZE 21134 . 21726) (MAXSIZE* 21730 . 21987) (MULTIPLE
21991 . 22182) (NNULLATOM 22186 . 22235) (OCCURS* 22239 . 22418) (
PATHANAL 22422 . 24353) (PICTURE* 24357 . 24512) (PICTUREOK 24516
 . 24642) (PREPROCESS 24646 . 25223) (QUAL 25227 . 25440) (QUALIFIEROK
25444 . 25746) (RECORDLIST 25750 . 25810) (RECORDNAME 25814 . 25872)
(SECTIONLIST 25876 . 25942) (SECTIONP 25946 . 26007) (SENTENCESCAN
26011 . 26816) (SONS 26820 . 27068) (VALUE* 27072 . 27250) (VC 27254
 . 27701) (VCG1 27705 . 28314) (ZAP 28318 . 28409)))))
STOP
```