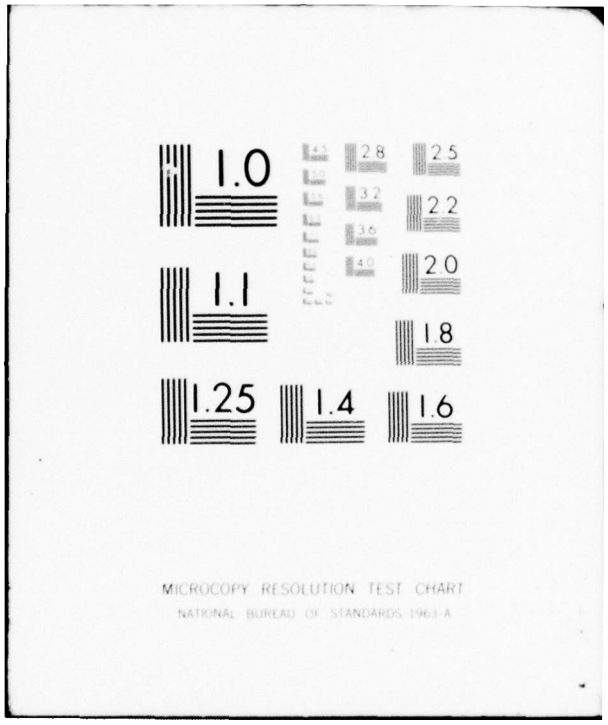
AD-A048 257 STANFORD RESEARCH INST MENLO PARK CALIF F/G 9/2
THE VERIFICATION OF COBOL PROGRAMS.(U)
JUN 75 L ROBINSON, M W GREEN, J M SPITZEN DAHC04-75-C-0011
UNCLASSIFIED NL

1 OF 2
AD
AD48257


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

BC

(9) TECHNICAL DOCUMENTARY REPORT.

(1)

U. S. ARMY COMPUTER SYSTEMS COMMAND

(6) THE VERIFICATION OF COBOL PROGRAMS.

INTERIM REPORT
SRI Project 3967

(10) Authors: L. Robinson,
M. W. Green
J. M. Spitzen

(11) 15 June 1975 (12) 99 p.

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

Prepared by
STANFORD RESEARCH INSTITUTE
333 Ravenswood Avenue
MENLO PARK, CALIFORNIA 94025

U. S. ARMY RESEARCH OFFICE Contract No. (15) DAHC 04-75-C-0011

DISTRIBUTION STATEMENT

Approved for public release. Distribution Unlimited.

Approved:

David R. Brown
David R. Brown, Director
Information Science Laboratory

John Manley Jr.
Jack Goldberg
Project Supervisor

332 500

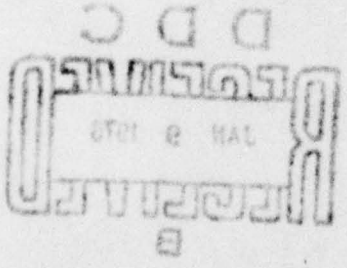
LB

DISPOSITION INSTRUCTIONS

Destroy this report when no longer needed. Do not return it to the originator.

DISCLAIMER

The findings of this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.



ABSTRACT

This report describes the progress of an investigation concerning the verification of COBOL programs. The report contains discussions of program verification, the COBOL language, and the role of structured programming in COBOL verification. The report also contains a presentation of a COBOL subset suitable for an experimental verification system--its syntax and semantics. The report also contains a discussion of the assertion language and rules of inference to be used in a COBOL verification system.

FOREWORD

This document was prepared under the authority of U.S. Army Research Office Contract No. DAHC 04-75-C-0011 in accordance with Part II, Article 4 of the contract, and was prepared by Stanford Research Institute for the U.S. Army Computer Systems Command. This report describes some preliminary results in an investigation concerning the verification of COBOL programs.

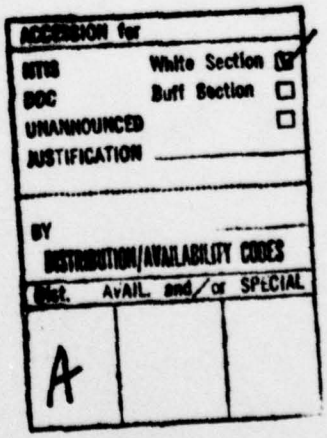


TABLE OF CONTENTS

1. Introduction

This report describes the progress of a project intended to study the issues involved in the verification of COBOL programs, and to produce some simple examples of verified programs in a selected subset of COBOL.

Given that program verification is useful in improving the reliability of programs, and that it is of great importance that COBOL programs be reliable (the vast majority of all programming is done in COBOL), it is certainly worthwhile to examine the feasibility of applying verification techniques to COBOL programs. One question is, "Why hasn't it been done sooner?" The answer lies in two factors:

(1) COBOL is a "real" language (i.e., designed for and used by a large community of users). Verification has only recently been applied to real languages, because of the relative newness of verification and because of the great complexity of real languages.

(2) Verification has, up to now, been practiced mainly by academicians, and academicians have a known distaste for COBOL.

This project is mainly a feasibility study, with some research and proof of concept. Once the major issues in COBOL verification are determined, we intend to illustrate what it means to verify a COBOL program (on a very small scale). The research involved is intended to extend current verification techniques to make them applicable to COBOL. This report is devoted mainly to a discussion of issues, and to a description of the techniques that we are developing.

The body of the report contains general motivational material describing the theory, observations, and general approach of the project. The three appendices contain descriptions of the particular results of the project so far--the syntax and semantics of the COBOL subset for verification.

2. Program Verification - Theory

The idea of program verification goes back as far as programming itself: it was first discussed by von Neumann and Goldstine (1). 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 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 (i.e., 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, for a given initial state of the program, what the final state will be. Often he will have a specification, which is a mapping from initial states to final states. It is not immediately obvious whether a program (a mapping from states to sequences of operations) and a specification (a mapping from states to states) are consistent. Consistency between a specification and a program is often called program correctness. Program verification is a set of techniques for proving this consistency. Floyd (2) 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 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 as well). The control operations, which do not in themselves affect the state, must also be axiomatized. Since a program may, in a small number of statements, describe a large (possibly infinite) sequence of operations, inductive assertions 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 formula in first order logic can be constructed whose validity is equivalent to the partial correctness of the program. This formula is called a verification condition. A software system that accepts as input the program to be verified (with input, output, and inductive assertions) is called a verification condition generator (3,4). Verification conditions can be proved by hand, or can serve as input to a deductive system, or automatic theorem prover, which attempts to generate a proof. Most 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 semi-automatic verification systems.

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 statements in the programming language into expressions in predicate calculus. Some verification condition generators are based on a particular semantic description of a language. A verification condition generator for PASCAL (London, Luckham, and Igarashi, 4) is based on the axiomatic description of PASCAL by Hoare and Worth (5).

A verification condition generator axiomatizes the control structures of the language, but properties of the data types of a language are often too complex to be incorporated into the verification conditions themselves. Verification conditions, especially in a high-level language, typically contain references to functions that axiomatize the data types of the language. The deductive system can prove formulae containing these functions either by invoking their definitions (if supplied) or by applying axioms (or high-level rules of inference) to make deductions. The first method works well for primitive recursive functions (Boyer and Moore, 6) but is extremely inefficient for more complex domains. Most verification systems, including the SRI system (Elspas et al., 3; Waldinger and Levitt, 7), use the second method. However, in this method all proofs may not be trusted if the axioms are wrong. One approach to this problem is to use high-level rules of inference to find a proof, and to check its validity using definitions and a proof checker (Boyer et al., 8). The proof checker would be used to substantiate the

validity of any instantiation of an axiom that is actually used in a proof. This may be easier than proving the most general form of the axiom from the definitions.

There are several areas that have not been addressed by the mainstream of program verification. The first is termination. This issue has been addressed by several researchers (3,9,10), and can be treated either together with or separately from the issue of partial correctness. Two other issues, run-time errors and validity of input data, are also important to formalize 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 (Sites, 11). Although these issues are important, they will not be considered during this contract, which must limit itself to the basic issues of partial correctness for COBOL programs.

3. Program Verification for Real Languages

COBOL is a member of the set of "real" programming languages, i.e., 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/1, and (to a lesser extent) Algol and LISP. The properties that make a programming language a real language unfortunately also serve to detract from the ease of verifying programs in that language. Most of these undesirable properties can be summed up under the term "lack of semantic cleanliness."

A language has a "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. The first reason is the size of the language. A real language is the incorporation of the special interests of many groups of users, whose interests are not always compatible. The result is often that large numbers of features are added on. The addition of these features not only complicates the semantics of the language, but often violates the spirit that motivated the initial conception of the language. PL/1 is a good example of this tendency. In a desire to overcome some of the difficulties of FORTRAN, COBOL, and Algol, the designers of PL/1 created something larger than any of its ancestors. Considered alone, the size of real languages is a major obstacle to verification. Second, most real languages must concede

syntactic generality in the interests of a fast implementation, either in 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 places (FORTRAN). Lack of syntactic generality makes the syntactic analysis phase of the verification system more difficult to implement. Third, most 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 to make 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 block will not affect the correctness of a COBOL program, but may affect the performance of that program. Fourth, 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 (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--"augumented" by the improvements. Another aspect of this problem is that most of the currently important languages got their start 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. Sturctured 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 by which the design of future programming languages will be guided. Lastly there is the problem that even if the semantics of a real language is clean, they are usually stated in natural language in a standards manual (12). A standards manual may be all right for programmers and language implementers, but it is certainly difficult for verification. If the standards people had some clean vision of a language in mind, they should have written down the formal semantics somewhere. The formal definition of PL/1 is such an attempt. The length of the formal definition of PL/1 is a commentary on our tools for specifying programming language semantics (e.g., VDL) and on the inherent semantic complexity of real languages.

Before solutions to these problems are considered, there is one major constraint to these solutions: the solutions must have minimum effect on the languages themselves. Manufacturers do not want to rewrite their compilers,

and users do not want to rewrite their programs. 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 useful on a more immediate basis.

The problem of language size has two aspects, syntactic, and semantic. When a language has syntactic complexity, 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 there exists more syntactic complexity than semantic, verification can be done on a program written in an internal form which is syntactically simple, i.e., 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 involves choosing a sublanguage that permits only the desired semantic features. Real languages differ in the extent to which subsets can be generated for them. If a language construct is necessary, but also semantically messy, there will be trouble in doing subsetting. This is precisely the trouble with the go to. It is clearly necessary in languages like FORTRAN and COBOL, but also permits the writing of programs with very messy control structures. The solution to this type of problem takes several forms:

(1) Try to change the language.

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

(3) Develop a preprecessor for the language, which permits desirable constructs in place of harmful ones.

It is the goal of this project to propose a subset of COBOL such that preprocessing need not take place. For more information, see Section 4, on structured programming and COBOL.

In the case of sacrificing syntactic generality for the speed of the compiler or the generated code, it is desirable to allow the verification system to process a language with more syntactic generality. To prevent the successful verification of programs that will not even compile, one then requires that all programs be run through the syntactic analysis phase of the compiler before verification. Thus the compiler can check the special

cases of the language, allowing the verification system's parser to be simpler (see Appendix I). Such a decision is made in this effort.

With regard to the features of a real language that are dependent on the hardware or the operating system, there are two strategies: either to axiomatize them or ignore them. Statements in COBOL's ENVIRONMENT DIVISION, and items like SYNCHRONIZED or the number of logical records per block, can be ignored since they do not affect the outcome of the program. Special kinds of file I/O and communication with the operating system can be axiomatized as properties of the abstract machine on which a program runs. The formal definition of a programming language involves specifying the instruction set of an abstract machine that runs the program, and specifying the interpreter that runs the program on the abstract machine.

The technology of program verification has ignored several issues that are essential in the verification of programs written in real languages. One reason for this phenomenon is that researchers in program verification are still having difficulty in applying program verification to toy languages (partly because verification is a comparatively new technique and because it is an extremely difficult one). These problems are also difficult in themselves. Among the problems 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 more often. Consideration of these items will appear in the section dealing with the semantics of COBOL data items.

Clean termination has been described in an earlier section. Because of the limited scope of this project this issue will not be dealt with at this time. Clean termination assumes the absence of run-time errors. However, such assumptions cannot always be made, as is the case in hardware and operating system errors and in situations where input data is invalid (see below). At some point such possibilities should be considered in efforts to verify programs in real language.

In verification the asumption is made that input data is valid (with respect to type, range of values, etc.). One of the greatest difficulties in assuming the reliability of programs in real languages is that such assumptions cannot be made. In other words it is a frequent occurrence that input data are faulty, and programs must be written to account for such situations. A real program will typically have several degraded modes of performance (without blowing up), depending on the severity of the error. Thus even if a single record is messed up, 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.

4. Structured Programming and COBOL

There is a growing interest in various techniques for increasing the "well-structuredness" of COBOL programs. This section discusses their impact on verification. The techniques fall into the categories of preprocessors and restrictions on the way in which COBOL programs are written.

The intent of these techniques is to simulate a block-structured language, in which control is nested. This kind of structure within a program makes the program easier to understand and debug. The conclusions are less certain for proof.

Let us first examine the preprocessors. Instead of the go to, they offer a set of control primitives such as do...while, if...then...else, case, and others. The intent is that such well-behaved control structures are easier to axiomatize than the go to and thus it would be easier to prove programs using only those constructs. This was the belief of Hoare (5), in his axiomatization of PASCAL. However, Knuth (13) in his paper on structured programming with the go to reports that go to's are surprisingly easy to axiomatize (see the appendix on the semantics of the COBOL subset). It is only necessary to put assertions at each label (and at PERFORM loops). Thus, there is no decrease per se, in the complexity of verification when go to's are removed from the language.

However, sturctured programming was intended to limit the complexity of the programs being written by reducing the average number of control paths per line of code. Structured programming also reduces the number of patterns of control paths by forcing the paths to be nested. Since the complexity of

Floyd verification is roughly proportional to the number of paths, it seems that (on the average) structured programs--whether written by preprocessor or by management fiat--are easier to verify than unstructured ones.

There is another sense in which the term "structured programming" is applicable. Structure can be gained by breaking large programs up into small, loosely coupled pieces. This is the modularity concept of Parnas (4), in which the change in a single design decision affects only one module. Unfortunately, the COBOL language itself does not provide facilities (such as flexible procedure calls or macros) for accomplishing this goal. In many cases management techniques are used to break up a large programming project into small, manageable pieces. One of the methods for accomplishing modularity is to hide the format of data structures within a single module. Since data structures (i.e., shared files) are precisely the means by which COBOL programs communicate, the format information for the data structures tends to be scattered over many programs. Thus, a change in file formats may require a lot of reprogramming, more than might be necessary if concepts of modularity were more visible in COBOL.

Decomposing a program into hierarchical levels of abstraction has been suggested (Dijkstra, 15) as a means for handling program complexity. Recently Robinson and Levitt (16) have proposed a method for formalizing a level of abstraction in a self-contained way, and for decomposing the proof of the large program into many small independent proofs, one for each level of abstraction. The applicability of this work to COBOL is perhaps a long way off, because the hierarchical method depends strongly on the notion of data abstraction. COBOL programs do not seem to have data structures that can be abstracted very easily. In spite of the tree-structured data in COBOL programs, all data structures seem to have one level of detail that is not hidden from parts of the program. Abstraction would not in this case lead to simple programs at higher levels. However, the problem bears further study.

5. Discussion of the COBOL Language

COBOL is a language of fairly simple control, but its data structures and operations are rich. The area of most immediate concern for verification is the elementary data item. All computation in COBOL is character-oriented. Numeric data items have pictures and sizes. Arithmetic operations must consider truncation and overflow with almost every operation. Even without the primitives STRING and UNSTRING, the manipulation of strings is inherent in each operation.

The most important feature of an elementary numeric data item is its PICTURE, a specification of how it would look if it were printed out. For example, a picture specification of 999 would print out a three-digit integer. The sign and decimal point information are also included in the specification. Although the decimal point in numeric items is implicit (remembered by the system but not stored with the item), the sign (if present) is encoded in one of the digits of the stored data item. A great deal of string processing can be performed by a simple assignment operation, because of the editing feature. There is a special type of data item called numeric edited, whose picture specification can contain additional information concerning inserted characters, zero suppresion, sign printing, and currency symbols. For example, a data item with a picture specification of $$$,$$$.99 would print out $10,000.00 when its contents are 10000 and $5.63 when its contents are 5.63. Notice that the comma disappears and the dollar sign moves over when the value of the item decreases. These features can be used to generate fancy reports, and can also create complexities with regard to verification. The editing must be axiomatized, and functions must be added to the assertion language in order to state properties of numeric edited data items. There are many data features that are not axiomatized by the subset provided by this project, e.g., string processing, table handling, sorting, and overlays.

COBOL is a language of input-output. There is sequential, random, indexed, and console I/O. Any verification system that deals with COBOL must handle I/O to some extent. This subset will handle console I/O and some very simplified versions of sequential I/O, enough to verify some elementary programs. We are using a method similar to Hoare's axiomatization of I/O in PASCAL (5). In it, a file is a sequence of values for the set of variables that constitute the input or output record. Each file has a pointer that designates the current record. Reading the file simply moves the pointer, while writing the file adds to the sequence and changes the pointer as well.

The records in COBOL are tree-structured, an attribute which presents a naming problem. Several elementary items may have the same local name, with the ambiguity resolved by different qualification statements. The CORRESPONDING option makes use of this feature. A COBOL verification system must incorporate the same naming mechanism that COBOL uses.

Several data features not incorporated in the subset are the REDEFINES and RENAMES options. REDEFINES allows a data item, either group or elementary, to have a different name and a different definition (i.e., set of picture specifications). It is like FORTRAN's COMMON statement, except that the sharing is done within one program. RENAMES allows the renaming of a sequence of elementary data items, but the same pictures are retained. It is analogous to the FORTRAN EQUIVALENCE statement. The REDEFINES option is much more difficult to handle, since it involves representation decisions in the machine, e.g., the number of characters contained in a group or elementary data item. These decisions also involve alignment and word boundaries, factors which vary depending on the implementation machine.

6. COBOL Subset for Verification

We have carefully examined the syntax and semantics of the COBOL language as defined by (12), and have arrived at a subset suitable for verification according to the criteria described in the previous sections of this report. The results of this research are described in the Appendices I, II, and III. Appendix I describes the syntax of the PROCEDURE DIVISION for the COBOL subset, the method (transduction grammars) for describing such syntax, the software system for manipulating these grammars, and the parsing program that uses them. Appendix II describes the syntax of the DATA DIVISION. Not all of the decisions have been made concerning the transductions for names and pictures of data items, so that the transductions are left out. Appendix III contains a discussion of the issues involved in the description of the semantics of COBOL statements and data types. This is a difficult problem, perhaps the most difficult of the project; only preliminary results have been shown here.

If one were to examine a list of the primitives that have been eliminated from the COBOL subset for verification, they could have been eliminated for one of two reasons:

(1) A primitive was considered to be undesirable for the purposes of verification.

(2) A primitive was considered to be reasonable for verification, but was not deemed "essential." Thus it was eliminated from this subset, which had to be kept small.

Very few constructs have been eliminated from the language for reason (1): the ALTER statement, the "abbreviated combined conditional" relational expression, the MOVE statement between group data items, and the REDEFINES and RENAMES statements. Even these features could be axiomatized, but with great difficulty.

The method for representing the COBOL grammar in the verification system is designed to allow extensions to the language at any time. It is predicted that further work in the project will call for the enlargement of the subset of COBOL handled by the verification system.

7. Assertion Language and Rules of Inference

The object of the assertion language is to allow a COBOL programmer to state any property of a COBOL program in an elegant way. This involves experimentation with many different COBOL programs to see what must be said and how to say it. At this time the assertion language design is in its very preliminary stages, and this section is a set of general guidelines that will motivate the final assertion language design.

Formulae in the assertion language must be handled by a general theorem-proving program, so that the syntactic basis for any assertion language must be first-order logic. The assertion language must deal with numeric quantities, so that arithmetic operators and relations are also included. Although sets do not occur in COBOL, they are useful in aggregating a multiplicity of items in assertions. Sequences appear in the axiomatization of files and strings, and are an otherwise useful structure. These general features should occur in any assertion language.

Instead of augmenting the syntax of the assertion language by adding language-dependent constructs, it is useful just to use functions and predicates to define these constructs. In order to perform deductions, axioms and definitions are used to describe properties of the functions and predicates. Axioms constitute high-level rules of inference and definitions can be viewed as substitution rules.

The particular functions used to describe the properties of COBOL are interesting. They fall into one of four categories:

(1) Type information. These functions tell whether an alphanumeric data item contains alphabetic or numeric data at a given time.

(2) Values of data items. Each numeric data item has a numeric value (real or integer) and a print value (character string).

(3) Naming information. The semantics of some COBOL operations depend on the data names above and below a data item in the tree-structured data definition.

(4) Operations on data items. Truncation, rounding, and editing of data items require special functions.

The enumeration and definition of these predicates and functions is now in progress.

8. Structure of Proposed Verification System

In our view of the problems of verification in real languages, we actually require the assistance of the compiler in the verification process. In addition, large parts of the verifier are table driven, so that certain changes in the COBOL subset will have a minimal effect on the programs comprising the verification system.

The proposed verification system is shown in Figure 1. In it, systems or processes (i.e., parts of the verification system) are denoted by ovals or circles. Documents or programs (i.e., the data that is processed by the verification system) are represented by rectangles. Knowledge encoded in system tables is represented by diamonds.

A program is first compiled by a standard COBOL compiler to check for syntax errors. Then user-supplied assertions are added to the program text, and the combined argument is fed to the parser of the verification system. Using the syntactic specifications for the language (the transduction grammar), the parser creates an internal form for the COBOL program. The verification condition generator takes the program in internal form and (using its knowledge of COBOL operations) produces the verification conditions. The verification conditions are then fed to the interactive deductive system, which attempts to produce a proof of the verification conditions (with the help of a human).

The scope of this project calls for the programming of the parser and verification condition generator. However, the most difficult issues are involved in deciding formal representation media for the items in the three diamonds, and for encoding the COBOL syntax and semantics using the representation media.

The system is being implemented on the PDP-10 at the Artificial Intelligence Center at Stanford Research Institute, using the INTERLISP programming environment. The system provides sophisticated interactive facilities for all phases of the programming process. The SRI facility is accessible through the ARPANET (address SRI-AI). Much of the documentation for the project is on-line at the same facility.

9. Conclusions

It is our feeling that we have uncovered some very interesting areas of study, and that COBOL verification is feasible and challenging. The level of effort does not permit as deep an examination of some of the issues as we had hoped, but this research provides a basis for further work.

The current status of the project can be summed up as follows:

(1) We have a thorough knowledge of the general issues of COBOL verification.

(2) We have decided on the syntax of the COBOL subset but have not yet finished axiomatizing it. However, a substantial amount of work has already been done.

(3) The parser has been written, and the verification condition generator has been sketched out.

(4) The documentation is adequate and up to date.

(5) Some sample COBOL programs have been studied, and assertions for them have been written.

(6) Except for the exact choice of auxiliary functions, the assertion language has been designed.

The following tasks remain to be done:

(1) Completion of semantic axiomatization, the choice of functions of the assertion language, and the rules of inference for the functions. These tasks are all related.

(2) Implementation of the verification condition generator. Given the completion of task (1), this is a relatively straightforward programming effort.

(3) More work on examples--both in writing assertions and generating hand-proofs. We will devote our attention to programs containing about 15-50 lines of PROCEDURE DIVISION.

(4) Completion of the study of structuring methods (including hierarchical methods) as applied to COBOL verification. At this time such efforts do not seem so fruitful as they did earlier. Perhaps we will have to devise slightly new techniques for partitioning the proofs of COBOL programs.

Several observations may be made:

(1) COBOL is an interesting language and is well designed.

(2) Structure and abstraction are not as promising as originally anticipated (see 4 above).

(3) We have been able to bring a surprising amount of technology to bear on the problems encountered.

The following problems either exist now or are anticipated:

(1) With the functions for editing and truncation, verification conditions may be longer than originally anticipated.

(2) It is taking more time than originally anticipated to arrive at a formal statement of COBOL semantics.

The following issues, although they will not be covered in the current effort, are important and deserve to be studied in future projects.

(1) Clean termination of COBOL programs.

(2) Graceful degradation in the presence of invalid data.

(3) The application of verification techniques to other areas related to the reliability of COBOL programs--e.g., testing, symbolic evaluation, and debugging.

It certainly seems as though the verification of COBOL programs is possible, and eventually may become cost effective.

References

1. 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).

2. R. W. Floyd, "Assigning Meanings to Programs," Proc. American Mathematical Society Symposium in Applied Mathematics, Vol. 19, pp. 19-31 (1967).

3. B. Elspas, K. N. Levitt, 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).

4. 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).

5. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica 2, 335-355 (1973).

6. R. Boyer and J. S. Moore, "Proving Theorems about LISP Functions," J.ACM 22, 1, pp. 129-144 (January 1975).

7. R. J. Waldinger and K. N. Levitt, "Reasoning about Programs," Artificial Intelligence 5, pp. 235-316 (1974).

8. R. Boyer, B. Elspas, R. E. Shostak, J. M. Spitzen, personal communication (1975).

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," Proc. International Conference on Reliable Software, 21-13 April 1975, International Hotel, 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. D. E. Knuth, "Structured Programming with go to Statements," Computing Surveys 6, 4, pp. 261-302 (December 1972).

14. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Comm. ACM 15, 12, pp. 1053-58 (December 1972).

15. E. W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," Report on a Conference on Software Engineering, Randall and Naur, eds., NATO (1968).

16. L. Robinson and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," SRI Report, submitted to Comm. ACM (January 1975).

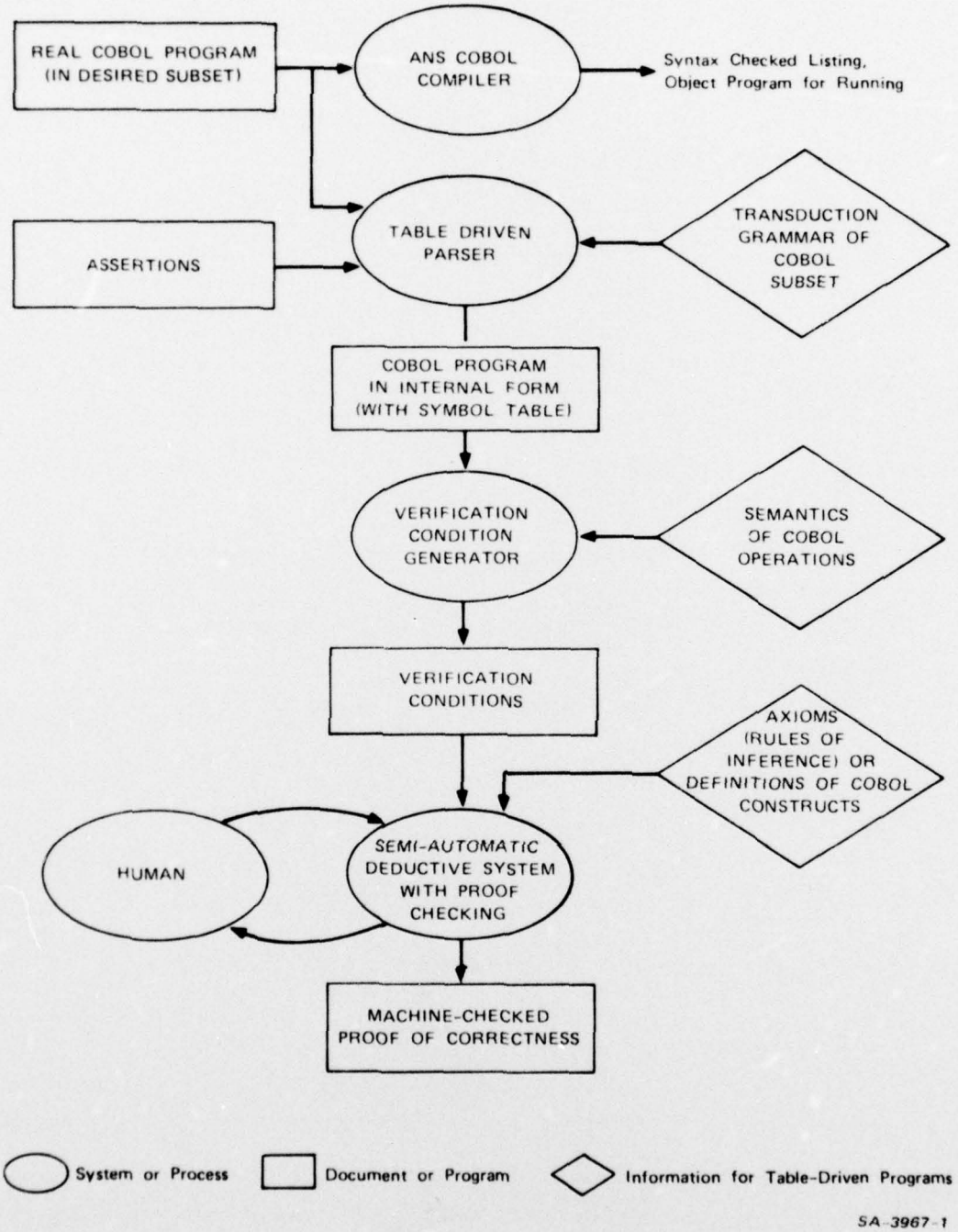


FIGURE 1 STRUCTURE OF THE COBOL VERIFIER

APPENDIX I

COBOL Language, Parsing, and Abstract Form

Jay M. Spitzen

1. Introduction

We intend to use a table-driven language transducer 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 import but with a uniform structure easily manipulated by a verification condition generator (the next phase of verification). Such a procedure is especially helpful in dealing with COBOL: this language has extensive syntactic complexities that often do not correspond to comparable semantic complexities. The point of the syntactic complexity of the language is to permit programmers to write in an expressive and natural format. While such a format is quite suitable for human consumption, it is inappropriate for the sorts of machine manipulation needed in verification, and it is consequently beneficial to translate to the syntactically much simpler abstract form that we have devised.

The correspondence between COBOL and Abstract COBOL 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. The transduction is a LISP program which computes the abstract form of the language fragment specified by the associated production. Thus we translate a COBOL program to abstract form 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 using transductions to obtain the parts of the desired Abstract COBOL program.

Our transduction grammar for a substantial subset of the constructs allowed in the COBOL procedure division, together with various parsing and grammar manipulating tools (described in Section 2), not only specifies the correspondence between COBOL and Abstract 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, 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$. Similarly, GO TO ... DEPENDING ON ... sentences are

expressed in Abstract COBOL by an equivalent set of IF and GO TO sentences. A detailed description of these correspondences, and of the primitives of Abstract COBOL, is given in Section 3.

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 purposes of the initial phase of the project activity, it will be a simple matter to extend the subset of COBOL that is accepted just by augmenting the CTG. Such extensions require no modification of the transducer.

2. The COBOL Language

A. Amendments to the Language

For the purpose of this project, we have designed a CTG for the COBOL procedure division which does not precisely correspond to the language described in the 1974 ANSI standard for COBOL (q.v. American National Standard Programming Language COBOL, American National Standards Institute, Standard Number X3.23-74). Our amendments are of two sorts and we now proceed to describe them in turn.

The first sort of amendment moderately extends the language. In the arithmetic sentences ADD, DIVIDE, MULTIPLY, and SUBTRACT, our CTG permits the arguments to be arbitrary arithmetic expressions rather than just

identifiers or literals. In the GO TO...DEPENDING ON... sentence, we similarly generalize the qualifying identifier so as to allow any arithmetic expression. In the PERFORM...AFTER... construct, we allow arbitrary nesting rather than a maximum of three levels as in X3.23. More importantly, the CTG specifies a generalized (but fully compatible) PERFORM statement which, for example, permits the construction

```
PERFORM procedure-name1 VARYING I FROM 1 BY 1 UNTIL I=10
                        5 TIMES
```

This construction is not allowed in X3.23 but is semantically consistent with it when given the meaning

```
PERFORM procedure-name1 VARYING I FROM 1 BY 1 UNTIL I=10
                        VARYING J FROM 1 BY 1 UNTIL J=5
```

where J is some new identifier not otherwise used in the program.

These extensions are permitted for a number of reasons. First, because they are semantically consistent with X3.23, it is no more difficult to verify programs written in the more general forms. Second, since the generalized forms are more syntactically natural (i.e. yield greater syntactic uniformity in the resulting language) than the original forms, the CTG is shorter and clearer than it would otherwise be. Finally, we could easily augment appropriate rules of the CTG to exclude these extensions if, for some reason, that was eventually found desirable. But, in any case, since the extensions are all compatible with X3.23,

the CTG does correctly specify the transduction into Abstract COBOL of standard language programs not employing the extensions.

Our second sort of amendment has consisted of subsetting the procedure division so that, in this limited initial effort, we can deal with a language of manageable proportions. On the other hand, we wish to include enough of COBOL to demonstrate the practicality of applying verification techniques to COBOL, as well as to begin to detail the techniques required. Thus we have tried to choose a group of verbs that is representative of COBOL and, moreover, is sufficient in scope to permit the writing and verification of some reasonable example programs. The technique of table-driven syntax transduction makes it quite easy to extend the subset with which we deal, and such extension would be a natural part of a continuation of the present effort.

The particular subset we have chosen includes the verbs ACCEPT*, ADD, COMPUTE, CLOSE*, DISPLAY, DIVIDE, GO, IF, MOVE, MULTIPLY, OPEN*, PERFORM, READ, STOP*, SUBTRACT, and WRITE. Asterisks indicate verbs for which the CTG allows only a subset of the alternative constructions in X3.23. We have excluded verbs dealing with string manipulation, table handling, merge and sort operations, error processing and debugging, complex file processing, interprocess

communications and multi-processing, and report generation. The verbs thus excluded are ALTER, DELETE, DISABLE, ENABLE, ENTER, EXIT, GENERATE, INITIATE, INSPECT, MERGE, RECEIVE, RELEASE, RETURN, REWRITE, SEARCH, SET, SORT, STRING, SUPPRESS, TERMINATE, UNSTRING, and USE. We have also excluded the 'abbreviated combined conditional' relational expression.

B. The Correspondence between COBOL and Abstract COBOL

The object of this section is to describe, in general terms, the correspondence between standard COBOL programs and their transduced versions in Abstract COBOL. Strictly speaking, the CTG as given in Appendix A is the definitive specification of this correspondence--it is both exact and procedural. However, the level of detail in Appendix A, together with the formal languages of transduction grammar and CLISP that are used, may be quite difficult for the uninitiated reader. Consequently we give a more tutorial presentation here.

The procedure division of a COBOL program consists of a number of labeled sections, each of which is made up of a number of labeled paragraphs. These paragraphs, in turn, are made up of a variable number of sentences. Alternatively, a program may omit the intermediate level (sections) and consist simply of a number of paragraphs. Both cases are represented in Abstract COBOL by lists of the

form

```
(PROCEDUREDIVISION$
        (SECTION$ section-name1
                (PARAGRAPH$ paragraph-namea
                        sentence sentence ...)
                ...
                (PARAGRAPH$ paragraph-namez
                        sentence sentence ...))
        ...
        (SECTION$ section-namen
                (PARAGRAPH$ paragraph-nameaa
                        sentence sentence ...)
                ...
                (PARAGRAPH$ paragraph-namezz
                        sentence sentence ...)))
```

We represent the case of a 'sectionless' program by taking n=1 and section-name1=NIL, i.e., by

```
(PROCEDUREDIVISION$
        (SECTION$ NIL
                (PARAGRAPH$ paragraph-namea
                        sentence sentence ...)
                ...
                (PARAGRAPH$ paragraph-nameb
                        sentence sentence ...)))
```

The sentences of Abstract COBOL serve the same function as the sentences of COBOL in that they serve as the building blocks of the language. Standard COBOL sentences are of two kinds--those represented in Abstract COBOL by a single internal sentence and those represented in Abstract COBOL by several internal sentences. The first sort of sentence is defined by the nonterminal 'sentence1' in the CTG. Such sentences are those using one of the following verbs: ACCEPT, CLOSE, GO, IF, PERFORM, READ, STOP, WRITE, ADD CORR, and SUBTRACT CORR. The second sort of sentence--comprising sentences with the verbs COMPUTE, DISPLAY, DIVIDE,

GO...DEPENDING, OPEN, MOVE, ADD, DIVIDE, MULTIPLY, or SUBTRACT--is represented in the CTG by the nonterminal 'sentence2'. To some extent there is a possible trade-off between the designation of the type of a COBOL sentence and the complexity of the associated internal semantic primitives. That is, forcing a sentence to be of the first type may require the use of more complex internal primitives than would transducing as a 'sentence2' to a list of internal sentences. To increase simplicity in the verification condition generator and COBOL axiomatization, we have therefore chosen to transduce to lists of simpler internal sentences where possible.

We now sketch the internal equivalents of the various COBOL sentences. Those derived from the nonterminal 'sentence1' are described first. Among these sentences, those with verbs ACCEPT, CLOSE, GO (the simple case), IF, READ, STOP, and WRITE are straightforwardly transduced. For example,

ACCEPT x FROM DAYTIME

becomes

(ACCEPT x DAYTIME)

where ACCEPT is considered, in Abstract COBOL, to be a semantically primitive function of two arguments. Similarly,

IF x+3<10; s1; ELSE s2.

becomes

(IF (LT$ (+ 3 x) 10) s1′ s2′)

where s1′ and s2′ are the Abstract COBOL equivalents of the external sentences s1 and s2.

Note that the CTG rules for condition and arithmetic expression yield the functional form (LT$ (+ 3 x) 10) for the COBOL infix expression x+3<10. The CTG translates any condition or arithmetic expression into a functional form employing only operators chosen from +, -, *, /, LT$, EQ$ or GT$, or the logical operators AND, OR, NOT, ISALPHABETIC$, and ISNUMERIC$.

There are two somewhat more complex cases. The first of these is the PERFORM statement. We analyze a PERFORM statement into three parts: the verb PERFORM, a body such as FUM THROUGH FUMBAR, and a list of controls. Each control is a qualifier such as 7 TIMES or AFTER J FROM 1 BY 3 UNTIL J IS GREATER THAN 15 and is analyzed into a keyword and a parameter. For example, the control 7 TIMES has keyword TIMES and parameter 7. The control AFTER J FROM 1 BY 3 UNTIL J IS GREATER THAN 15 has keyword VARYING and parameter (J 1 3 (GT$ J 15)). COBOL allows two other sorts of control; these are 'UNTIL condition' and a defaulted control (as in 'PERFORM FUM'). The first of these has keyword UNTIL and as parameter the transduction of the condition. The second has keyword ONCE and parameter NIL. Suppose the external form of a PERFORM statement is

```
PERFORM body control(1) ... control(n).
```

For each i between 1 and n, let kcontrol(i) be the keyword associated with control(i) and let pcontrol(i) be its parameter. Then we transduce to the internal equivalent

```
(PERFORM kcontrol(n)
         (PERFORM kcontrol(n-1) ... pcontrol(n-1) NIL)
         pcontrol(n)
         NIL)
```

Thus the semantic primitive PERFORM in Abstract COBOL takes four arguments: a control keyword, a transduction of the body which is either

```
(DO$ procedure-name1 procedure-name2)
```

(the simple case) or the transduction of the nested inner PERFORM, a control parameter list, and a final argument which is--at present--NIL. As a more complex example, consider the sentence

```
PERFORM PAR1 VARYING I FROM 1 BY 1 UNTIL I=10
             7 TIMES
             UNTIL X<10.
```

In internal form, this will be represented by

```
(PERFORM VARYING
         (PERFORM TIMES
                  (PERFORM UNTIL
                           (DO$ PAR1 PAR1)
                           (LT$ X 10)
                           NIL)
                  7
                  NIL)
         (I 1 1 (EQ$ I 10))
         NIL)
```

Observe that when we proceed, in planned project work, to use the transduced program as an input to a verification condition generator, an additional item of information will

be needed for each iteration. This will be an inductive invariant which describes the logical behavior of the iteration body, and it will be recorded in the final argument position of the corresponding PERFORM. Ideally, one would prefer to have a verification system synthesize such an invariant on the basis of the program text, but it is not possible to do so in real programs given the present state of the art of verification.

The remaining derivatives of 'sentence1' are those using ADD CORR and SUBTRACT CORR. These translate to calls on the semantic primitives ADDCORRESPONDING$ and SUBTRACTCORRESPONDING$ with four arguments: the two apparent subjects of the COBOL sentence, either ROUNDED or NIL as specified by the external sentence, and a transduction of the COBOL imperative sentence that is to be executed if a SIZE ERROR occurs.

We now describe the translation of derivatives of 'sentence2' in the grammar. Recall that these generally translate to several internal sentences. To represent the COMPUTE and other arithmetic sentences, we introduce the semantic functions SET$ and SETROUNDED$. Each is a function of three arguments: the target of the operation, the source expression, and an error sentence analogous to the third ADDCORRESPONDING$ argument as described above. For example, consider the COBOL sentence

```
ADD x,y TO z,w ROUNDED.
```

We translate this to the two internal sentences

```
(SET$ z (+ (+ z y) x) NIL)
(SETROUNDED$ w (+ (+ w y) x) NIL)
```

Other arithmetic verbs are handled in the same fashion, with the CTG transductions creating the proper functional form source expression. Observe that in the example a SIZE ERROR imperative statement is omitted; if it were present then its transduction would appear in the proper argument positions in each of the resulting internal sentences. We handle MOVE in the same way, translating

```
MOVE x TO y.
```

to

```
(SET$ y x NIL)
```

For the MOVE CORRESPONDING statement, we introduce the semantic primitive MOVECORRESPONDING$ and translate, for example,

```
MOVE CORR a OF x TO b OF y.
```

to the internal form

```
(MOVECORRESPONDING$ (OF y (b)) (OF x (a)))
```

where the OF subexpressions are the internal renditions of COBOL qualifications.

The internal primitive DISPLAY is similar to ACCEPT described above. However, since the COBOL language allows DISPLAY to take a list of arguments, we transduce to a list of internal DISPLAYs, e.g.,

```
DISPLAY x,y,z UPON PRINTER.
```

becomes

```
(DISPLAY x PRINTER)
(DISPLAY y PRINTER)
(DISPLAY z PRINTER)
```

OPEN is translated in a similar way, using the three internal primitives OPENINPUT$, OPENOUTPUT$, and OPENBOTH$. For example,

```
OPEN I-O file1,file2.
```

becomes

```
(OPENBOTH$ file1)
(OPENBOTH$ file2)
```

Finally, we describe the transduction of GO...DEPENDING sentences. In general, such a sentence has the form

```
GO TO n1,n2,... DEPENDING ON expression.
```

and is translated as though it had been the sentence

```
IF expression=1; GO TO n1; ELSE
IF expression=2; GO TO n2; ELSE
...
```

which, rendered in Abstract COBOL, is

```
(IF (EQ$ expression 1)
    (GO n1)
    (IF (EQ$ expression 2)
        (GO n2)
        ...).
```

3. Interactive Facilities

We have developed a variety of interactive facilities to support the construction of CTGs and the subsequent parsing of COBOL programs. The system we describe is

written in the LISP programming language and runs under INTERLISP (q.v. INTERLISP Reference Manual by Warren Teitelman, Xerox Palo Alto Research Center).

The two basic functions used to create a CTG are PUTRULES and PUTTRANS. They are both variadic functions whose first argument is a nonterminal of the grammar and whose subsequent arguments are, respectively, the production rules and transductions for the nonterminal. When a nonterminal is initially used as first argument to PUTRULES or PUTTRANS, it is appended to the list NONTERMS of nonterminals thus far in the grammar. To distinguish this case, PUTRULES returns the nonterminal as its result; otherwise it returns NIL. The first nonterminal introduced becomes the root symbol of the grammar (e.g., the nonterminal 'proceduredivision' in the CTG of Appendix A).

Once a nonterminal and some corresponding <production, transduction> pairs have been specified in this way, adjustments to the grammar may be made by using PUTRULES and PUTTRANS to add additional alternatives for the nonterminal (or for other nonterminals), and by using the INTERLISP editing facilities to modify the alternatives then in effect. In particular, EDITV(NONTERMS) will allow the user to modify the list of nonterminals and EDITP(nt) will allow the user to modify the productions and transductions of a particular nonterminal.

The grammar (or any part of it) may be listed in a readable format (as in Appendices A and B) by calling PRINTGRAMMAR with any subset of NONTERMS as argument. Appendix A contains such a listing for the subset of the COBOL procedure division we have selected. Each nonterminal is printed along with a list of <production, transduction> pairs--one for each alternative. We have adopted the convention that lower case symbols denote nonterminals while upper case symbols and delimiters denote terminals. Also, note that the transductions are printed in the CLISP conversational dialect of LISP for increased conciseness and readability. (In this dialect, described in detail in Chapter 23 of the INTERLISP Reference Manual, angle brackets ('<' and '>') denote the list consisting of the bracketed elements. Thus <A B <C>> is equivalent to (LIST A B (LIST C)). However, an exclamation point indicates that the following element is to be inserted as a segment, e.g. <! A B ! C> is equivalent to (APPEND A (LIST B) C). Other notational innovations of CLISP that we use freely are apostrophe (') to quote the symbol or form that it precedes and colon and double colon as infix operators. X:I, where I is an integer denotes the Ith element of the list X; X::I denotes the Ith tail of X.)

Once the grammar has been refined to the user's satisfaction, it may be saved in a symbolic file for subsequent reference by the function call

SAVEGRAM(filename), which will also sort the nonterminals into alphabetic order (except for the root symbol which remains the first element of NONTERMS). Prior to this call, the user may also wish to sort the alternatives for each nonterminal into lexicographical order (based on the productions of the alternatives). This is done by the call SORTRULES(NONTERMS).

When the grammar is completed, the system may be used to transduce COBOL programs within the COBOL subset that has been defined. There are two functions available for this purpose--PURIFY and ABSTRACT. The first of these automatically transforms the grammar to an equivalent one that contains no erasing rules. This is important because the many optional words in the COBOL language lead to erasing nonterminals in the grammar (e.g., 'at' and 'is' in Appendix A). However, our parser has been designed to deal only with grammars without erasing rules; this permits a simpler and more efficient parser than would otherwise be possible. Consequently, a 'purification' process is needed to obtain a grammar acceptable to the parser. The effect of this process on a CTG may be seen by comparing Appendices A and B. For example, the nonterminal 'sentence1' has nine alternatives in the original grammar but requires twenty-three in the purified grammar to make up for the absence of erasing rules. A purified grammar may be saved with SAVEGRAM as described in the previous paragraph.

Once these preliminaries are complete, it is possible to parse a COBOL program. The user must enter the program into the LISP environment and then invoke the function ABSTRACT providing two arguments--the program and the function COBOLTOKENFN. The program is then parsed and transduced and the resulting value of ABSTRACT is the translated program in Abstract COBOL. Appendix C contains an example of this process: part 1 is a simple COBOL program and part 3 is the abstract form of the program. Part 2, included here for completeness but usually of no interest to a user, shows the parse tree which is constructed from the input program prior to the invocation of the transductions of the CTG.

Finally, let us describe the use of COBOLTOKENFN by ABSTRACT. The reader will observe that no rules are given for three nonterminals of the CTG--'symbol', 'number', and 'string'. This is because they correspond to the basic lexical symbols, numeric constants, and textual constants permitted in COBOL which are, naturally, much too numerous to be listed explicitly. Instead, as each lexical token of an input program is read by the parser, COBOLTOKENFN is invoked to check whether it is a symbol, number, or string. If so, the appropriate rule alternatives are added dynamically to these nonterminals so that parsing may proceed successfully.

Appendix A. COBOL Transduction Grammar

```
proceduredivision

=  PROCEDURE DIVISION . paragraphs
   (<'PROCEDUREDIVISION$ <'SECTION$ NIL ! T4>>)

=  PROCEDURE DIVISION . sections
   (<'PROCEDUREDIVISION$ ! T4>)

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

argument

=  expressions
   (T1)

=  expressions connector expression
   ((if T2 NEQ ('BY)
        then <! T1 T3> elseif T1::1 then (HELP
                                    "Error in reduction to
 argument.")
        else <T3 ! T1>))

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

at

=
   (NIL)

=  AT
   (NIL)

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

classcondition

=  ALPHABETIC
   ('ISALPHABETIC)

=  NUMERIC
   ('ISNUMERIC)

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

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)

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

conditionname

= symbol
  (T1)

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

connector

= BY
  (T1)

= FROM
  (T1)

= INTO
  (T1)

= TO
  (T1)

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

corresponding

= CORR
  (NIL)

= CORRESPONDING
  (NIL)

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

corrop

= ADD
  ('ADDCORRESPONDING$)

= SUBTRACT
  ('SUBTRACTCORRESPONDING$)

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

divideарguments

= expression BY expression
  (<T1 T3>)
```

```
= expression INTO expression
  (<T3 T1>)

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

else

= ELSE
  (NIL)

= OTHERWISE
  (NIL)

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

elseclause

=
  ('NEXT)

= semi else NEXT SENTENCE
  ('NEXT)

= semi else sentence
  (T3)

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

endcondition

=
  (NIL)

= ; at END sentence
  (T4)

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

errorcondition

=
  (NIL)

= ; on SIZE ERROR sentence
  (T5)

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

expression

= expression + expression2
  (<T2 T1 T3>)
```

```
= expression2
  (T1)

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

expression2

= expression2 * expression3
  (<T2 T1 T3>)

= expression2 / expression3
  (<T2 T1 T3>)

= expression3
  (T1)

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

expression3

= expression3 ** expression4
  (<T2 T1 T3>)

= expression4
  (T1)

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

expression4

= ( expression )
  (T2)

= + expression4
  (T2)

= - expression4
  (<T1 0 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>)

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

identifier

= qualification
  ((if (NLISTP T1)
       then T1 elseif T1:1='OF and T1:3=NIL then T1:2 else
       (HELP 'Error% in% reduction% to% identifier.)))

= qualification ( subscripts )
  (<'OF (if (NLISTP T1)
            then T1 elseif T1:1='OF and T1:3=NIL then T1:2
 else
            (HELP 'Error% in% reduction% to% identifier.))
    T3>)

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

identifiers

= identifier
  (<T1>)
```

```
= identifier , identifiers
  (<T1 ' T3>)

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

indexname

= symbol
  (T1)

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

iotype

= INPUT
  ('OPENINPUT$)

= IO
  ('OPENBOTH$)

= OUTPUT
  ('OPENOUTPUT$)

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

is

=
  (NIL)

= IS
  (NIL)

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

mnemonicname

= symbol
  (T1)

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

move

= MOVE
  ('SET$)

= MOVE corresponding
  ('MOVECORRESPONDING$)

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

```
of

=  IN
   (NIL)

=  OF
   (NIL)

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

on

=
   (NIL)

=  ON
   (NIL)

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

operator

=  ADD
   ('+)

=  DIVIDE
   ('/)

=  MULTIPLY
   ('*)

=  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
  (<T2 T1>)

= expression TIMES
  (<T2 T1>)

= varying expression FROM expression BY expression UNTIL
 condition
  (<T1 <T2 T4 T6 T8>>)

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

performcontrols

=
  (NIL)

= performcontrol performcontrols
  (<T1 ! T2>)

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

procedurename

= symbol
  (T1)

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

procedurenames

= procedurename
  (<T1>)
```

```
= procedurename , procedurenames
  (<T1 ! T3>)

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

qualification

= symbol
  (<'OF T1 NIL>)

= symbol of qualification
  (<T3:1 T3:2 <! T3:3 T1>>)

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

readtarget

=
  (NIL)

= INTO identifier
  (T2)

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

record

=
  (NIL)

= RECORD
  (NIL)

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

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

=  ACCEPT identifier source
   (<T1 T2 T3>)

=  CLOSE filenames
   (<T1 ! T3>)

=  GO to procedurename
   (<T1 T3>)

=  IF condition thenclause elseclause
   (<T1 T2 T3 T4>)

=  PERFORM performbody performcontrols
   ((if T3 then (for (X R_T2)
```

```
                    in
                    (REVERSE T3)
                    do R_ <'PERFORM X:1 R X:2 NIL> finally
                    (RETURN R))
          else <'PERFORM ' (ONCE$)
          T2 NIL NIL>))

=  READ filename record readtarget endcondition
   (<T1 T2 T4 T5>)

=  STOP RUN
   (T1)

=  WRITE recordname writesource
   (<T1 T2 T3>)

=  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>))

=  DISPLAY identifiers target
   ((for X in T2 collect <T1 X T3>))

=  DIVIDE divideargumentS GIVING computetarget1 REMAINDER
 identifier
errorcondition
   (<<! T4 <'/ ! T2> T7> <'SET$ T6 <NIL T2:1 <'* T4:2 T2:2>>
 T7>>)

=  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 <T1 X T2 NIL>))

=  operator arguments GIVING computetarget errorcondition
   ((for X in T4 collect <! X (for (Y (R_ T2:-1))
                                   in
                                   (REVERSE T2)
```

```
                                        ::1 do R_ <T1 Y R>
 finally
                                        (RETURN R))
        T5>))

=  operator expressions connector computetarget
 errorcondition
   ((for X in T2 join
         (for Y in T4 collect <! Y <T1 Y:2 X> T5>)))

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

sentences

=  sentence1 .
   (<T1>)

=  sentence1 . sentences
   (<T1 ! T3>)

=  sentence2 .
   (T1)

=  sentence2 . sentences
   (<! T1 ! T3>)

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

signcondition

=  NEGATIVE
   (' (GT$ 0))

=  NOT NEGATIVE
   (' (LTQ$ 0))

=  NOT POSITIVE
   (' (GTQ$ 0))

=  NOT ZERO
   (' (NEQ$ 0))

=  POSITIVE
   (' (LT$ 0))

=  ZERO
   (' (EQ$ 0))

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

simplecondition
```

```
= conditionname
  (T1)

= expression is relationoperator expression
  (<T3 T1 T4>)

= expression is signcondition
  (<!! T3 T1>)

= identifier is classcondition
  (<T3 T1>)

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

source

= FROM DATE
  (T2)

= FROM DAY
  (T2)

= FROM TIME
  (T2)

= FROM mnemonicname
  (T2)

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

subscripts

= expression
  (<T1>)

= expression , subscripts
  (<T1 ! T3>)

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

target

=
  (NIL)

= UPON mnemonicname
  (T2)

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

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)

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

writesource

=
   (NIL)
```

= FROM identifier
(T2)

Appendix B. COBOL Non-erasing Transduction Grammar

```
proceduredivision

=  PROCEDURE DIVISION . paragraphs
   (<´PROCEDUREDIVISION$ <´SECTION$ NIL ! T4>>)

=  PROCEDURE DIVISION . sections
   (<´PROCEDUREDIVISION$ ! T4>)

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

argument

=  expressions
   (T1)

=  expressions connector expression
   ((if T2 NEQ (´BY)
        then <! T1 T3> elseif T1::1 then (HELP
                                    "Error in reduction to
 argument.")
        else <T3 ! T1>))

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

at

=  AT
   (NIL)

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

classcondition

=  ALPHABETIC
   (´ISALPHABETIC)

=  NUMERIC
   (´ISNUMERIC)

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

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)
```

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

conditionname

=  symbol
   (T1)

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

connector

=  BY
   (T1)

=  FROM
   (T1)

=  INTO
   (T1)

=  TO
   (T1)

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

corresponding

=  CORR
   (NIL)

=  CORRESPONDING
   (NIL)

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

corrop

=  ADD
   ('ADDCORRESPONDING$)

=  SUBTRACT
   ('SUBTRACTCORRESPONDING$)

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

divideargument s

=  expression BY expression
   (<T1 T3>)

=  expression INTO expression
   (<T3 T1>)
```

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

else

=  ELSE
   (NIL)

=  OTHERWISE
   (NIL)

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

elseclause

=  semi else NEXT SENTENCE
   ('NEXT)

=  semi else sentence
   (T3)

=  else NEXT SENTENCE
   ('NEXT)

=  else sentence
   (T2)

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

endcondition

=  ; at END sentence
   (T4)

=  ; END sentence
   (T3)

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

errorcondition

=  ; on SIZE ERROR sentence
   (T5)

=  ; SIZE ERROR sentence
   (T4)

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

expression

=  expression + expression2
   (<T2 T1 T3>)
```

```
= expression2
  (T1)

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

expression2

= expression2 * expression3
  (<T2 T1 T3>)

= expression2 / expression3
  (<T2 T1 T3>)

= expression3
  (T1)

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

expression3

= expression3 ** expression4
  (<T2 T1 T3>)

= expression4
  (T1)

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

expression4

= ( expression )
  (T2)

= + expression4
  (T2)

= - expression4
  (<T1 0 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>)

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

identifier

= qualification
  ((if (NLISTP T1)
       then T1 elseif T1:1='OF and T1:3=NIL then T1:2 else
       (HELP 'Error% in% reduction% to% identifier.)))

= qualification ( subscripts )
  (<'OF (if (NLISTP T1)
            then T1 elseif T1:1='OF and T1:3=NIL then T1:2
 else
            (HELP 'Error% in% reduction% to% identifier.))
    T3>)

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

identifiers

= identifier
  (<T1>)
```

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

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

indexname

= symbol
  (T1)

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

iotype

= INPUT
  ('OPENINPUT$)

= IO
  ('OPENBOTH$)

= OUTPUT
  ('OPENOUTPUT$)

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

is

= IS
  (NIL)

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

mnemonicname

= symbol
  (T1)

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

move

= MOVE
  ('SET$)

= MOVE corresponding
  ('MOVECORRESPONDING$)

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

of

= IN
  (NIL)
```

```
= OF
  (NIL)

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

on

= ON
  (NIL)

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

operator

= ADD
  ('+)

= DIVIDE
  ('/)

= MULTIPLY
  ('*)

= 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
   (<T2 T1>)

=  expression TIMES
   (<T2 T1>)

=  varying expression FROM expression BY expression UNTIL
 condition
   (<T1 <T2 T4 T6 T8>>)

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

performcontrols

=  performcontrol performcontrols
   (<T1 ! T2>)

=  performcontrol
   (<T1>)

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

procedurename

=  symbol
   (T1)

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

procedurenames

=  procedurename
   (<T1>)

=  procedurename , procedurenames
   (<T1 ! T3>)

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

qualification
```

```
= symbol
  (<'OF T1 NIL>)

= symbol of qualification
  (<T3:1 T3:2 <! T3:3 T1>>)

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

readtarget

= INTO identifier
  (T2)

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

record

= RECORD
  (NIL)

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

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$)

=  EQUAL
   ('EQ$)

=  GREATER
   ('GT$)

=  LESS
   ('LT$)

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

rounded

=  ROUNDED
   (T1)

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

section

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

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

sectionname

=  symbol
   (T1)

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

```
sections

=  section
   (<T1>)

=  section sections
   (<T1 ! T2>)

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

semi

=  ;
   (NIL)

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

sentence

=  sentence1
   (T1)

=  sentence2
   ((if T1::1 then <'DO$ ! T1> else T1:1))

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

sentence1

=  ACCEPT identifier source
   (<T1 T2 T3>)

=  CLOSE filenames
   (<T1 ! T3>)

=  GO to procedurename
   (<T1 T3>)

=  IF condition thenclause elseclause
   (<T1 T2 T3 T4>)

=  PERFORM performbody performcontrols
   ((if T3 then (for (X R_T2)
                     in
                     (REVERSE T3)
                     do R_ <'PERFORM X:1 R X:2 NIL> finally
                     (RETURN R))
        else <'PERFORM ' (ONCE$)
        T2 NIL NIL>))

=  READ filename record readtarget endcondition
   (<T1 T2 T4 T5>)
```

```
=  STOP RUN
   (T1)

=  WRITE recordname writesource
   (<T1 T2 T3>)

=  corrop corresponding identifier connector identifier
 rounded
errorcondition
   (<T1 T3 T5 T6 T7>)

=  GO procedurename
   (<T1 T2>)

=  IF condition thenclause
   (<T1 T2 T3 'NEXT >)

=  PERFORM performbody
   ((if NIL then (for (X R_T2)
                      in
                      (REVERSE NIL)
                      do R_ <'PERFORM X:1 R X:2 NIL> finally
                      (RETURN R))
        else <'PERFORM ' (ONCE$)
        T2 NIL NIL>))

=  READ filename readtarget endcondition
   (<T1 T2 T3 T4>)

=  READ filename record endcondition
   (<T1 T2 NIL T4>)

=  READ filename endcondition
   (<T1 T2 NIL T3>)

=  READ filename record readtarget
   (<T1 T2 T4 NIL>)

=  READ filename readtarget
   (<T1 T2 T3 NIL>)

=  READ filename record
   (<T1 T2 NIL NIL>)

=  READ filename
   (<T1 T2 NIL NIL>)

=  WRITE recordname
   (<T1 T2 NIL>)

=  corrop corresponding identifier connector identifier
 errorcondition
   (<T1 T3 T5 NIL T6>)
```

```
=  corrop corresponding identifier connector identifier
 rounded
   (<T1 T3 T5 T6 NIL>)

=  corrop corresponding identifier connector identifier
   (<T1 T3 T5 NIL NIL>)

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

sentence2

=  COMPUTE computetarget = expression errorcondition
   ((for X in T2 collect <! X T4 T5>))

=  DISPLAY identifiers target
   ((for X in T2 collect <T1 X T3>))

=  DIVIDE divideaarguments GIVING computetarget1 REMAINDER
 identifier
errorcondition
   (<<! T4 <'/ ! T2> T7> <'SET$ T6 <NIL T2:1 <'* T4:2 T2:2>>
 T7>>)

=  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 <T1 X T2 NIL>))

=  operator arguments GIVING computetarget errorcondition
   ((for X in T4 collect <! X (for (Y (R_ T2:-1))
                                   in
                                   (REVERSE T2)
                                   ::1 do R_ <T1 Y R>
 finally
                                   (RETURN R))
         T5>))

=  operator expressions connector computetarget
 errorcondition
   ((for X in T2 join
         (for Y in T4 collect <! Y <T1 Y:2 X> T5>)))

=  COMPUTE computetarget = expression
   ((for X in T2 collect <! X T4 NIL>))
```

```
=  DISPLAY identifiers
   ((for X in T2 collect <T1 X NIL>))

=  DIVIDE divideareguments GIVING computetarget1 REMAINDER
 identifier
   (<<! T4 <´/ ! T2> NIL> <´SET$ T6 <NIL T2:1 <´* T4:2
 T2:2>> NIL>>)

=  GO procedurenames DEPENDING on expression
   ((for I to (LENGTH T2)
         collect
         (<´IF <´EQ$ T5 I> <´GO (CAR (NTH T2 I))
           > ´ NEXT >)))

=  GO to procedurenames DEPENDING expression
   ((for I to (LENGTH T3)
         collect
         (<´IF <´EQ$ T5 I> <´GO (CAR (NTH T3 I))
           > ´ NEXT >)))

=  GO procedurenames DEPENDING expression
   ((for I to (LENGTH T2)
         collect
         (<´IF <´EQ$ T4 I> <´GO (CAR (NTH T2 I))
           > ´ NEXT >)))

=  operator arguments GIVING computetarget
   ((for X in T4 collect <! X (for (Y (R_ T2:-1))
                                   in
                                   (REVERSE T2)
                                   ::1 do R_ <T1 Y R>
 finally
                                   (RETURN R))
         NIL>))

=  operator expressions connector computetarget
   ((for X in T2 join
         (for Y in T4 collect <! Y <T1 Y:2 X> NIL>)))

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

sentences

=  sentence1 .
   (<T1>)

=  sentence1 . sentences
   (<T1 ! T3>)

=  sentence2 .
   (T1)
```

```
= sentence2 . sentences
  (<! T1 ! T3>)

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

signcondition

= NEGATIVE
  (' (GT$ 0))

= NOT NEGATIVE
  (' (LTQ$ 0))

= NOT POSITIVE
  (' (GTQ$ 0))

= NOT ZERO
  (' (NEQ$ 0))

= POSITIVE
  (' (LT$ 0))

= ZERO
  (' (EQ$ 0))

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

simplecondition

= conditionname
  (T1)

= expression is relationoperator expression
  (<T3 T1 T4>)

= expression is signcondition
  (<!! T3 T1>)

= identifier is classcondition
  (<T3 T1>)

= expression relationoperator expression
  (<T2 T1 T3>)

= expression signcondition
  (<!! T2 T1>)

= identifier classcondition
  (<T2 T1>)

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

```
source

=  FROM DATE
   (T2)

=  FROM DAY
   (T2)

=  FROM TIME
   (T2)

=  FROM mnemonicname
   (T2)

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

subscripts

=  expression
   (<T1>)

=  expression , subscripts
   (<T1 ! T3>)

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

target

=  UPON mnemonicname
   (T2)

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

than

=  THAN
   (NIL)

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

thenclause

=  NEXT SENTENCE
   ('NEXT)

=  semi sentence
   (T2)

=  sentence
   (T1)

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

```
thru

= THROUGH
  (NIL)

= THRU
  (NIL)

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

to

= TO
  (NIL)

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

varying

= AFTER
  ('VARYING)

= VARYING
  ('VARYING)

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

writesource

= FROM identifier
  (T2)

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

Appendix C. A Sample Transduction

1. A COBOL Program

```
PROCEDURE DIVISION .
START-HERE .
    OPEN INPUT ACNT-FILE .
    MOVE ZERO TO STORE .
READ-IT .
    READ ACNT-FILE ; AT END GO TO END-IT .
    ADD 1 TO STORE .
COMPARE .
    IF ACNT-NO EQUAL STORE GO READ-IT .
    DISPLAY STORE .
    IF STORE EQUAL 99 STOP RUN .
    ADD 1 TO STORE .
    GO COMPARE .
END-IT .
    COMPUTE STORE = STORE + 1 .
    IF STORE IS GREATER THAN 99 ; STOP RUN .
    DISPLAY STORE .
    GO TO END-IT .
```

2. The Corresponding Parse Tree

```
((root . 1)
 ((proceduredivision . 1)
  PROCEDURE DIVISION %.
  ((paragraphs . 2)
   ((paragraph . 1)
    ((paragraphname . 1) ((symbol . 1) START-HERE))
    %.
    ((sentences . 4)
     ((sentence2 . 5)
      OPEN
      ((iotype . 1) INPUT)
      ((filenames . 1)
       ((filename . 1) ((symbol . 2) ACNT-FILE))))
     %.
     ((sentences . 3)
      ((sentence2 . 6)
       ((move . 1) MOVE)
       ((expression . 2)
        ((expression2 . 3)
         ((expression3 . 2) ((expression4 . 4) ZERO))))
       TO
       ((identifiers . 1)
        ((identifier . 1)
         ((qualification . 1) ((symbol . 3) STORE)))))
      %.)))
   ((paragraphs . 2)
```

```
((paragraph . 1)
 ((paragraphname . 1) ((symbol . 4) READ-IT))
 %.
 ((sentences . 2)
  ((sentence1 . 15)
   READ
   ((filename . 1) ((symbol . 2) ACNT-FILE))
   ((endcondition . 1)
    ;
    ((at . 1) AT)
    END
    ((sentence . 1)
     ((sentence1 . 3)
      GO
      ((to . 1) TO)
      ((procedurename . 1) ((symbol . 5) END-IT))))))
  %.
  ((sentences . 3)
   ((sentence2 . 16)
    ((operator . 1) ADD)
    ((expressions . 1)
     ((expression . 2)
      ((expression2 . 3)
       ((expression3 . 2)
        ((expression4 . 8) ((number . 1) 1))))))
    ((connector . 4) TO)
    ((computetarget . 1)
     ((computetarget1 . 1)
      ((identifier . 1)
       ((qualification . 1) ((symbol . 3) STORE))))))
   %.)))
((paragraphs . 2)
 ((paragraph . 1)
  ((paragraphname . 1) ((symbol . 6) COMPARE))
  %.
  ((sentences . 2)
   ((sentence1 . 11)
    IF
    ((condition . 2)
     ((condition2 . 2)
      ((condition3 . 2)
       ((condition4 . 2)
        ((simplecondition . 5)
         ((expression . 2)
          ((expression2 . 3)
           ((expression3 . 2)
            ((expression4 . 7)
             ((identifier . 1)
              ((qualification . 1)
               ((symbol . 7) ACNT-NO)))))))
         ((relationoperator . 2)
          ((relationoperator2 . 7) EQUAL))
         ((expression . 2)
```

```
          ((expression2 . 3)
           ((expression3 . 2)
            ((expression4 . 7)
             ((identifier . 1)
              ((qualification . 1)
               ((symbol . 3) STORE))))))))))))
  ((thenclause . 3)
   ((sentence . 1)
    ((sentence1 . 10)
     GO
     ((procedurename . 1) ((symbol . 4) READ-IT))))))
%.
((sentences . 4)
 ((sentence2 . 10)
  DISPLAY
  ((identifiers . 1)
   ((identifier . 1)
    ((qualification . 1) ((symbol . 3) STORE)))))
 %.
 ((sentences . 2)
  ((sentence1 . 11)
   IF
   ((condition . 2)
    ((condition2 . 2)
     ((condition3 . 2)
      ((condition4 . 2)
       ((simplecondition . 5)
        ((expression . 2)
         ((expression2 . 3)
          ((expression3 . 2)
           ((expression4 . 7)
            ((identifier . 1)
             ((qualification . 1)
              ((symbol . 3) STORE)))))))
        ((relationoperator . 2)
         ((relationoperator2 . 7) EQUAL))
        ((expression . 2)
         ((expression2 . 3)
          ((expression3 . 2)
           ((expression4 . 8)
            ((number . 2) 99)))))))))))
   ((thenclause . 3)
    ((sentence . 1) ((sentence1 . 7) STOP RUN))))
  %.
  ((sentences . 4)
   ((sentence2 . 16)
    ((operator . 1) ADD)
    ((expressions . 1)
     ((expression . 2)
      ((expression2 . 3)
       ((expression3 . 2)
        ((expression4 . 8) ((number . 1) 1))))))
    ((connector . 4) TO)
```

```
          ((computetarget . 1)
           ((computetarget1 . 1)
            ((identifier . 1)
             ((qualification . 1) ((symbol . 3) STORE))))))
         %.
         ((sentences . 1)
          ((sentence1 . 10)
           GO
           ((procedurename . 1) ((symbol . 6) COMPARE)))
          %.))))))
    ((paragraphs . 1)
     ((paragraph . 1)
      ((paragraphname . 1) ((symbol . 5) END-IT))
      %.
      ((sentences . 4)
       ((sentence2 . 9)
        COMPUTE
        ((computetarget . 1)
         ((computetarget1 . 1)
          ((identifier . 1)
           ((qualification . 1) ((symbol . 3) STORE)))))
        =
        ((expression . 1)
         ((expression . 2)
          ((expression2 . 3)
           ((expression3 . 2)
            ((expression4 . 7)
             ((identifier . 1)
              ((qualification . 1) ((symbol . 3)
STORE)))))))
         +
         ((expression2 . 3)
          ((expression3 . 2)
           ((expression4 . 8) ((number . 1) 1))))))
       %.
       ((sentences . 2)
        ((sentence1 . 11)
         IF
         ((condition . 2)
          ((condition2 . 2)
           ((condition3 . 2)
            ((condition4 . 2)
             ((simplecondition . 2)
              ((expression . 2)
               ((expression2 . 3)
                ((expression3 . 2)
                 ((expression4 . 7)
                  ((identifier . 1)
                   ((qualification . 1)
                    ((symbol . 3) STORE)))))))
              ((is . 1) IS)
              ((relationoperator . 2)
               ((relationoperator2 . 5)
```

```
                    GREATER
                    ((than . 1) THAN)))
                  ((expression . 2)
                   ((expression2 . 3)
                    ((expression3 . 2)
                     ((expression4 . 8)
                      ((number . 2) 99))))))))))
           ((thenclause . 2)
            ((semi . 1) ;)
            ((sentence . 1) ((sentence1 . 7) STOP RUN))))
          %.
          ((sentences . 4)
           ((sentence2 . 10)
            DISPLAY
            ((identifiers . 1)
             ((identifier . 1)
              ((qualification . 1) ((symbol . 3) STORE)))))
           %.
           ((sentences . 1)
            ((sentence1 . 3)
             GO
             ((to . 1) TO)
             ((procedurename . 1) ((symbol . 5) END-IT)))
            %.))))))))))
RPAD)
```

3. The Corresponding Abstract Form

```
(PROCEDUREDIVISION$
 (SECTION$ NIL
        (PARAGRAPH$ START-HERE
                    (OPENINPUT$ ACNT-FILE)
                    (SET$ STORE 0 NIL))
        (PARAGRAPH$ READ-IT
                    (READ ACNT-FILE NIL (GO END-IT))
                    (SET$ STORE (+ STORE 1) NIL))
        (PARAGRAPH$ COMPARE
                    (IF (EQ$ ACNT-NO STORE) (GO READ-IT)
NEXT)
                    (DISPLAY STORE NIL)
                    (IF (EQ$ STORE 99) STOP NEXT)
                    (SET$ STORE (+ STORE 1) NIL)
                    (GO COMPARE))
        (PARAGRAPH$ END-IT
                    (SET$ STORE (+ STORE 1) NIL)
                    (IF (GT$ STORE 99) STOP NEXT)
                    (DISPLAY STORE NIL)
                    (GO END-IT))))
```

APPENDIX II

Syntax of the COBOL DATA DIVISION

L. Robinson

This document contains the syntax of the DATA DIVISION of the COBOL subset for verification. As is the case for the PROCEDURE DIVISION, the language is described as a transduction grammar. At this point in time, the transductions for the DATA DIVISION grammar have not been included. The objective of the transductions of the PROCEDURE DIVISION is to create a COBOL program in abstract form. The transductions of the DATA DIVISION can be used to construct a symbol table to be employed by the COBOL verification system.

The DATA DIVISION is divided into two parts, the FILE SECTION and the WORKING-STORAGE section. The FILE SECTION contains the information on files used by the program, and a description of the data records associated with the file. A data record contains the names and picture specifications (i.e., the declarations) of variables used in the program. The WORKING-STORAGE SECTION is used to declare the program variables not contained in data records of files. Variables in the WORKING-STORAGE SECTION may be declared individually or grouped into data records.

A file declaration contains several options, of which only a few are included in the subset. LABEL and DATA RECORD options are included, while BLOCK, RECORD, VALUE OF, LINAGE, CODE-SET, and REPORT are eliminated. BLOCK, RECORD, VALUE-OF, and CODE-SET are items of value to the implementing machine. LINAGE and REPORT are used by the report module of COBOL, none of whose primitives are part of the subset.

Record descriptions are tree-structured. A record description entry can designate a group item, in which case it contains a level number and a name, or an elementary item, in which case its picture, etc., are also described. If the value of an elementary item corresponds to a condition-name, a level-number of 88 is used together with a description of the values that signify the condition. A working-storage variable declared individually has a level number of 77. The first name in a data record description must have a level number of 01. For elementary or group items, any two-digit level number (except 01, 66, 77, or 88) may be used.

A data description entry characterizes an elementary data item. It consists of a level number followed by the name of the item or FILLER (if the item is not to be referenced at the elementary level by the program) and a list of options. We include the options PICTURE, JUSTIFIED,

and VALUE. Excluded are REDEFINES, USAGE, SIGN, OCCURS, SYNCHRONIZED, and BLANK ZERO. REDEFINES and OCCURS have not been axiomatized, and would require an enlargement of the subset. USAGE will only be DISPLAY for this subset, so it was eliminated (since that is the default). SIGN and BLANK ZERO can be handled by numeric editing. JUSTIFIED provides for the alignment of characters in an alphanumeric item when data items are moved to it. VALUE performs initialization of an elementary data item.

PICTURE specifications are the most complicated (and perhaps the most interesting) part of the DATA DIVISION grammar. There are three types of pictures, with the picture type determining the type of the data item. Alphabetic items may contain letters and spaces. Alphanumeric items may contain any printable characters. Numeric items contain fixed decimal or integer values. The picture specification may indicate that the data item is edited, in which case changes are made in order to print out the data item. Editing can take two forms--insertion and zero suppression. In insertion, extra characters are inserted between digits in the edited item. The nature of the insertion may depend of the value of the item. In zero suppression, leading zeros (and intervening insertion characters) to the left of the decimal point are replaced by spaces, asterisks, or spaces followed by either plus, minus, or currency sign. In PICTURE specifications, the kind of

editing is described by the sequence of characters involved. Since there are many possible character combinations (corresponding to the kinds of editing to be done), the grammar for the picture specifications is difficult indeed.

Appendix A. Grammar for the Data Division

```
data-division
=  DATA DIVISION .
=  DATA DIVISION . file-section
=  DATA DIVISION . file-section working-storage-section
=  DATA DIVISION . working-storage-section

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

$character
=  $
=  embeddedcharacter

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

$string
=  $character $string
=  $string

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

$string1
=  $string
=  $string decimalpoint
=  $string decimalpoint $string
=  $string pstring impliedpoint
=  decimalpoint $string
=  impliedpoint pstring $string

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

*character
=  *
=  embeddedcharacter

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

*string
=  *character
=  *character *string

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

9character
=  9
=  embeddedcharacter

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

```
9string
=  9character
=  9character 9string

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

creditdebit
=  +
=  -
=  CR
=  DB

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

currencyinsertion
=  $string editstring
=  $string1

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

data-description
=  88 symbol semi value ranges .
=  level-number data-name .
=  level-number data-name picture justification value-clause


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

data-description1
=  symbol picture justification value-clause

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

data-descriptions
=  data-description .
=  data-description . data-descriptions

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

data-name
=  FILLER
=  symbol

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

data-record
=
=  semi DATA record symbols

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

decimalpoint
=  .
```

```
= V

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

editstring
= editstring2
= pstring
= pstring impliedpoint

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

editstring1
= editstring2
= impliedpoint pstring 9string
= pstring 9string

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

editstring2
= 9string
= 9string decimalpoint
= 9string decimalpoint 9string
= 9string pstring
= 9string pstring impliedpoint
= decimalpoint 9string

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

embeddedcharacter
= 0
= ,
= /
= B

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

file-descriptor
= FD symbol label-record .
= FD symbol label-record data-record . record-descriptions

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

file-descriptors
= file-descriptor
= file-descriptor file-descriptors

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

file-section
= FILE SECTION .
= FILE SECTION . file-descriptors
```

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

impliedpoint
=
=  V

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

initialpart
=  $
=  $ +
=  $ -
=  + $
=  - $

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

is
=
=  IS

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

just
=  JUST
=  JUSTIFIED

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

justification
=
=  semi just
=  semi just RIGHT

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

label-record
=  semi LABEL record OMITTED
=  semi LABEL record STANDARD

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

literal
=  number
=  string

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

minuscharacter
=  -
=  embeddedcharacter
```

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

minusstring
=  minuscharacter
=  minuscharacter minusstring

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

numericedited
=  currencyinsertion
=  currencyinsertion creditdebit
=  initialpart numericedited1
=  numericedited1
=  numericedited1 creditdebit
=  sign currencyinsertion
=  signinsertion

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

numericedited1
=  editstring1
=  zerosuppression

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

picture
=
=  semi picture-word is picture-spec

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

picture-word
=  PIC
=  PICTURE

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

pluscharacter
=  +
=  embeddedcharacter

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

plusstring
=  pluscharacter
=  pluscharacter plusstring

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

pstring
=  P
=  P pstring
```

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

range
=  literal
=  literal thru literal

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

ranges
=  range
=  range , ranges

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

record
=  RECORD
=  RECORD IS
=  RECORDS
=  RECORDS ARE

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

record-description
=  01 symbol . data-descriptions

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

record-descriptions
=  record-description .
=  record-description . record-descriptions

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

semi
=
=  ;

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

sign
=  +
=  -

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

signinsertion
=  $ signstring editstring
=  signstring editstring
=  signstring1

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

```
signstring
=  minusstring
=  plusstring

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

signstring1
=  decimalpoint signstring
=  impliedpoint pstring signstring
=  minusstring decimalpoint minusstring
=  plusstring decimalpoint plusstring
=  signstring
=  signstring decimalpoint
=  signstring pstring impliedpoint

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

suppressstring
=  *string
=  zstring

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

suppressstring1
=  *string decimalpoint *string
=  decimalpoint suppressstring
=  impliedpoint pstring suppressstring
=  suppressstring
=  suppressstring decimalpoint
=  suppressstring pstring impliedpoint
=  zstring decimalpoint zstring

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

symbols
=  symbol
=  symbol symbols

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

thru
=  THROUGH
=  THRU

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

value
=  VALUE
=  VALUE IS
=  VALUES
=  VALUES ARE
```

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

value-clause
=
=  semi VALUE IS literal
=  semi VALUE literal

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

working-storage-list
=  77 data-description1 .
=  77 data-description1 . working-storage-list
=  record-description .
=  record-description . working-storage-list

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

working-storage-section
=  WORKING-STORAGE SECTION .
=  WORKING-STORAGE SECTION . working-storage-list

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

zcharacter
=  Z
=  embeddedcharacter

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

zerosuppression
=  suppressstring editstring
=  suppressstring1

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

zstring
=  zcharacter
=  zcharacter zstring

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

APPENDIX III

M. W. Green

Axiomatization of COBOL Semantics

This section reports on preliminary work toward an axiomatic representation of COBOL semantics. The aim is to describe an adequate, but somewhat restricted, subset of COBOL in such a way that automatic or semi-automatic generation of program verification conditions is facilitated. To a considerable extent we have been guided by Hoare's axiomatization of the language PASCAL [1]. However, COBOL is in some respects a much more complex language than PASCAL, so that some additional notational and meta-linguistic conveniences had to be improvised to describe the effect of certain COBOL statements.

The COBOL language is described in the ANSI Report [2] by a collection of syntatic forms accompanied by informal or "prose" specifications of the effect of each language statement. In interpreting this document we have noticed several instances where a restriction or a relaxation of the allowed language expressions would be helpful in formulating useful program verification conditions. Where these situations arise, we have arbitrarily chosen to use the most convenient interpretation (or restriction). In particular, we should mention

1. There are several instances where the description of COBOL syntax (as given in the ANSI report) seems unnecessarily restrictive. For example, the GOTO...DEPENDING ON [id] statement could just as well accept an integer-valued arithmetic expression (or even a COMPUTE...) instead of a simple identifier. Where we could see no reason for observing this sort of language restriction we have omitted it from the axiomatization of the version of COBOL that actually will be used in program proving. If a compiler forbids the more relaxed syntax, then the program is not well formed and proof of correctness is not an issue.

2. There are instances where a COBOL statement is considered to be too dangerous or too difficult to cope with in program verification. Examples are the ALTER verb and the MOVE statement applied to group data-items. In the former case, the possibly intricate variations in flow of program control are very difficult to handle. In the latter case, the predicates associated with all of the possible consequences of an unformatted transfer of alphanumeric data are extremely complex. For such reasons we will often omit some COBOL statement from the axiomatic description (forbid them) or restrict the generality of others.

3. Where the syntatic correctness of an allowed COBOL statement is clearly checkable by a compiler, we will assume correctness on the part of the compiler. Furthermore we assume that certain run-time checks that detect operands of inappropriate type will be compiled into the executable code. This means, for example, that we need not adjoin predicates to an ADD statement which assert that the arguments are numeric quantities. The consequences of a run-time error in data-type may be handled in at least two sensible ways. The first would attach an implicit ON ERROR GOTO... to each statement where such a situation could occur. We choose a simpler alternative in which these errors signify non-termination of the program. This is consistent with Hoare's treatment of PASCAL wherein $P\{S\}Q$ is satisfied if S diverges.

4. COBOL is a language with a rather weak notion of data-type, and a very elaborate collection of conversion rules. Other algebraic languages make do with a few standard internal representations for data-types and a few permissible high-level coercion rules such as INTEGER + REAL = REAL to preserve integrity of data-type. In these languages the explicit rounding or tuncation of numeric quantities to conform with non-standard internal representation must usually be accomplished by extra arithmetic manipulations expressed as high-level language statements. In the latter respect COBOL is peculiar (but not unique), because a simple assignment statement need not

preserve numeric equality between the sending and receiving values. For this reason we need some method of expressing the effect of transmitting the value of an elementary COBOL data-item to a receiving identifier. The route we have taken is to express these conversion rules as functions (without side-effects) that accept values and PICTURES as arguments and return values equivalent to the COBOL conversion rules. (See details below.)

Illustrative Examples

1. MOVE

The COBOL MOVE statement is the analog of the assignment statement in other high-level languages. For the most primitive form of this statement, MOVE x TO y; the corresponding PASCAL or ALGOL equivalent is y:= x. The Hoare axiomatization of this statement would be,

$$P_x^y\{\text{MOVE x TO y}\}P$$

where the notation P_x^y denotes the predicate derived from P by substitution of the value of x for y in P. Informally, if P is true after execution of {MOVE x TO y} then P_x^y is also true. Now the MOVE statement, in addition to having several variational forms, may also modify data so that $x \neq y$ after a MOVE. This occurs whenever x suffers an editing operation on being transferred to location y. All such editing operations may be described by functions having no side-effects such as $\text{Edit}(x,\text{pic}_y)$. Here, Edit is a function of two arguments, the value of x and the PICTURE corresponding to the variable y. The internal details of the function Edit implement the conversion rules described by the COBOL report and the function definition of Edit can serve to define the semantics of the conversion process.

In statements that manipulate arithmetic quantities, COBOL provides the option of truncating or rounding values that might not be accommodated to full precision in the receiving picture. Truncation is the normal default operation, but rounding can be forced by the use of the ROUNDED modifier in most arithmetic operations. The effects of truncation or rounding may also be described by editing functions, for example,

$$\text{Edittrunc}(x,\text{pic}_y) \text{ and } \text{Editround}(x,\text{pic}_y)$$

with equally precise internally defined semantics. In the following discussion

we will use the function Edit as a generic name for the conversion function. When a ROUNDED modifier appears in a COBOL statement it should be understood that Editround is to be used instead of Edit etc.

Standard COBOL permits a MOVE of non-elementary (group) items via an unformatted block-transfer of alphanumeric information. To avoid this dangerous programming practice, we make a restriction that MOVE x to y where x and y are group items is permitted only when x and y have identical picture-structure as defined in the DATA DIVISION. With this constraint all MOVE statements can be decomposed into MOVEs of elementary items.

Observing these conventions the axiom for the MOVE of an elementary data-item becomes

$$P^{y}_{E}\{\text{MOVE x TO y}\}P, \text{ where } E = \text{Edit}(x, \text{pic}_y)$$

The alternative form, MOVE x to a, b, ...is transduced by the CTG (see Appendix I) to a sequence of simple MOVES and therefore does not require separate axiomatization. However, to explicate this notion, we introduce the rule

$$P^{a,b,\ldots}_{E_1,E_2,\ldots}$$

$$E_1 = \text{Edit}(x, \text{pic}_a),\ E_2 = \text{Edit}(x, \text{pic}_b)\ \ldots$$

Now, according to Section 5.15.4 of the COBOL standard, value substitutions are to be carried out in sequence rather than "simultaneously." Thus, the statements

MOVE i to j, a(j)

MOVE i to a(j), j

should have different effects. Consequently, we will observe the convention that the notation

$$P^{a,b\ldots}_{E_1,E_2\ldots}$$

stands for the expression resulting from first substituting E_1 for $\underline{a}$ in P, then substituting E_2 for b in the derived expression, etc. This differs from the interpretation found in Hoare [1], where simultaneous substition was the rule.

The third variant of the MOVE statement, namely MOVE CORRESPONDING x TO y, involves group data-items. In the internal representation of the data division of a COBOL program there will be a form of symbol table that provides a unique address (name) for each elementary data-item. However this information is kept, an equivalent unique name for each elementary item can be specified by forming the ordered list of identifiers (i.e., qualifiers) proceeding from the name of the data item upward through each level of data subdivision to the 01 level. For example in the data structure

```
01  RECORD
    02  BAZ
        05  A
        05  B
```

the namelist of B is (B, BAZ, RECORD), and this is entirely equivalent to the specification in COBOL syntax, B IN BAZ IN RECORD.

Definition:

Two elementary items are CORRESPONDING with respect to id_1 and id_2 if $id_1 \neq id_2$ and the namelist of the first item up to but not including id_1 is identical with the namelist of the second item up to but not including id_2. Let Z be the set of ordered pairs

$$\{x_1y_1,\ x_2y_2 \ldots | x_iy_i \text{ CORRESPONDING in } X,Y\}$$

and $E_1 = \text{Edit}(x_1, \text{pic}_{y_1}) \ldots$

then the rule

$$\frac{\forall (x_iy_i) \in Z:\ P^{y_i}_{E_i}\{\text{MOVE } x_i \text{ TO } y_i\}P}{P^{y_1 \ldots}_{E_1 \ldots}\{\text{MOVE CORRESPONDING X TO Y}\}P}$$

gives the semantic interpretation in this form of the MOVE statement.

2. GOTO

The GOTO statement has two variants.

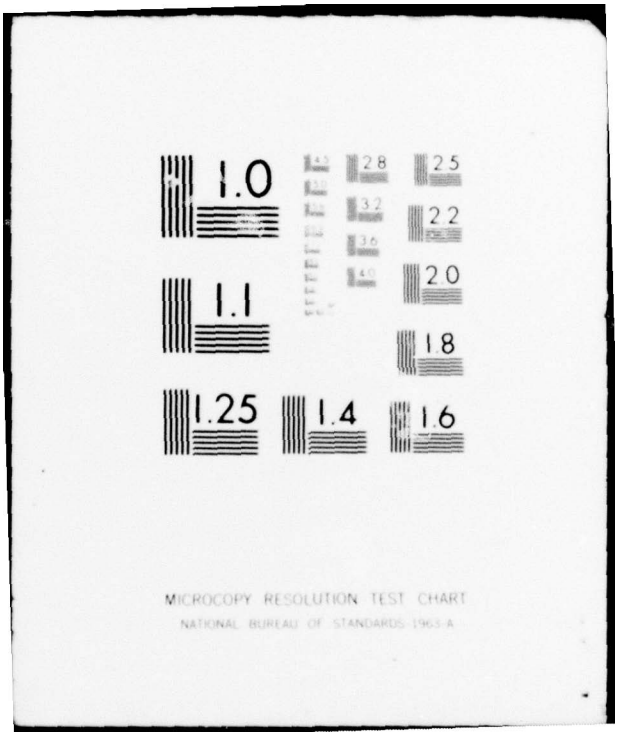
GOTO procedure-name,

GOTO procedure-name$_1$,[procedure-name$_2$]...
DEPENDING ON identifier.

AD-A048 257 STANFORD RESEARCH INST MENLO PARK CALIF F/G 9/2
THE VERIFICATION OF COBOL PROGRAMS.(U)
JUN 75 L ROBINSON, M W GREEN, J M SPITZEN DAHC04-75-C-0011
UNCLASSIFIED NL

2 OF 2
AD
AD48257

END
DATE
FILMED
1 -78
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

A suitable axiomatic treatment of the GOTO is given in Knuth[3]. For COBOL programs, we need the following rules. To each procedure-name L in the program that is the target or possible target of a GOTO we must provide a logical assertion predicate P(L) that must be true whenever flow of program control reaches L. Then

$$P(L)\ \{\text{GOTO L}\}\ \underline{\text{false}}$$

and the rule of inference

$$\frac{P(L)\ \{\text{body}\}Q}{P(L)\ \{\text{L: body}\}Q}$$

gives the appropriate condition that the GOTO must satisfy. Here body represents the statements belonging to a procedure-name.

For the second version of the GOTO statement, which resembles the ALGOL switch construct, the informal semantics are that the identifier is evaluated to an integer i and control is transferred to the ith procedure in the procedure-name list. If i is not an integer in the range 1 to n (n is the number of procedures in the name list) then the statement has no effect, i.e., control "falls through" to the next COBOL statement.

We handle this construct by developing the DEPENDING ON conditional into a set of equivalent IF statements during the transduction phase so that no separate axiomatization is required.

3. IF-ELSE

The syntatic form of this COBOL statement is

$$\text{IF condition}\ \left\{\begin{matrix}\text{statement}_1\\ \text{NEXT SENTENCE}\end{matrix}\right\}\ \left\{\begin{matrix}\text{ELSE statement}_2\\ \text{ELSE NEXT SENTENCE}\end{matrix}\right\}$$

Here some restrictions that existed in earlier versions of COBOL have been relaxed in the present ANSI standards to permit statement_1 and statement_2 to be of either imperative or conditional type. If we interpret the phrase NEXT SENTENCE as an imperative statement having no effect, then its axiomitization is simply

$$P\ \{\text{NEXT SENTENCE}\}\ P$$

and if S_1 and S_2 stand for permissible statements including NEXT STATEMENT then

$$\frac{P \wedge \text{condition } \{S_1\}Q,\ P \wedge \neg \text{condition } \{S_2\}Q}{P\{\text{IF condition } S_1 \text{ ELSE } S_2\}Q}$$

is the appropriate rule of inference for the IF-ELSE statement.

4. *ADD*

The ADD statement in COBOL with its several variations and its optional error exit is perhaps the most syntactically complex arithmetic statement to be found in any high-level language. Its axiomatization is fairly straightforward, however, having much the same form as that of the MOVE statement. (In fact, the semantic primitives *SET$* and *SETROUNDED$* represent both in Abstract COBOL.) In the most primitive form

ADD x TO y [ROUNDED]

we have, by analogy with MOVE,

$$P^{y}_{E}\ \{\text{ADD x TO y}\}\ P,\quad E = \text{Edit } (x+y, pic_y)$$

where Edit should be replaced by Editround if the ROUNDED modifier is employed. The more general form

ADD x,y,z ... TO u,v,w ... ,

we currently expand into multiple internal statements so the latter form needs no separate axiomatization. The variant

ADD x,y,z ...GIVING w [ROUNDED]

has a similar rule, namely

$$P^{w}_{E}\ \{\text{ADD x,y,z ...GIVING w}\}\ P,\quad E = \text{Edit } (x+y+z \ldots, pic_w)\ .$$

Here also, the appearance of a list of variables in the place of w would be expanded into separate internal statements. A third variant,

ADD CORRESPONDING X TO Y [ROUNDED],

leads to an axiom set very similar to that of the MOVE CORRESPONDING statement (see 1 above). That is,

$$Z = \{x_i y_i,\ x_2 y_2 \ldots | x_i y_i \text{ CORRESPONDING in X,Y}\}$$

$$E = \text{Edit } (x_1 + y_1, \text{pic}_{y_1}) \ldots$$

$$\frac{\forall (x_i y_i) \in Z:\ P_{E_i}^{y_i} \{\text{ADD } x_i \text{ TO } y_i\} P}{P_{E_1}^{y_1} \ldots \{\text{ADD CORRESPONDING X TO Y}\} P}$$

In each of the variants of the ADD statement, an optional clause [ON SIZE ERROR imperative statement] may be attached to take appropriate action if numeric overflow or underflow conditions arise in the computation. Therefore, we must consider the family of statements of which

ADD x TO y; ON SIZE ERROR S.

is typical (where S is some imperative statement). This statement requires several axioms. Let "Sum-fits(y,x+y)" be the assertion that the result of the computation "x+y" fits in location y. Then one correct rule of inference is

$$\frac{P\{\text{ADD x TO y}\}Q}{P \,\&\, \text{Sum-fits}(y,x+y)\{\text{ADD x TO y; ON SIZE ERROR S}\}Q}$$

i.e., in the absence of an error, the error condition is superfluous. Next, suppose that Sum-fits(y,x+y) is false. Then a complete axiomatization must distinguish two cases. The first case is that the error is detected before y is modified. Then we may use the inference rule:

$$\frac{\text{Early-detection}(y,x+y) \,\&\, (P\{S\}Q) \,\&\, \neg\, \text{Sum-fits}(y,x+y)}{P\{\text{ADD x TO y; ON SIZE ERROR S}\}Q}$$

The second case is that an error is detected after y has been modified. A complete axiomatization must then account for the execution of S in the modified environment. We intend, in our present work, to make the simplifying assumption that this case does not arise.

References

1. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL" Acta Informatica 2, 335-355 (1973).

2. ANSI X3.23 - 1974 American National Standard COBOL Report.

3. D. E. Knuth, "Structured Programming with GOTO Statements," Computing Surveys 6, 4, pp. 261-302 (December 1974).