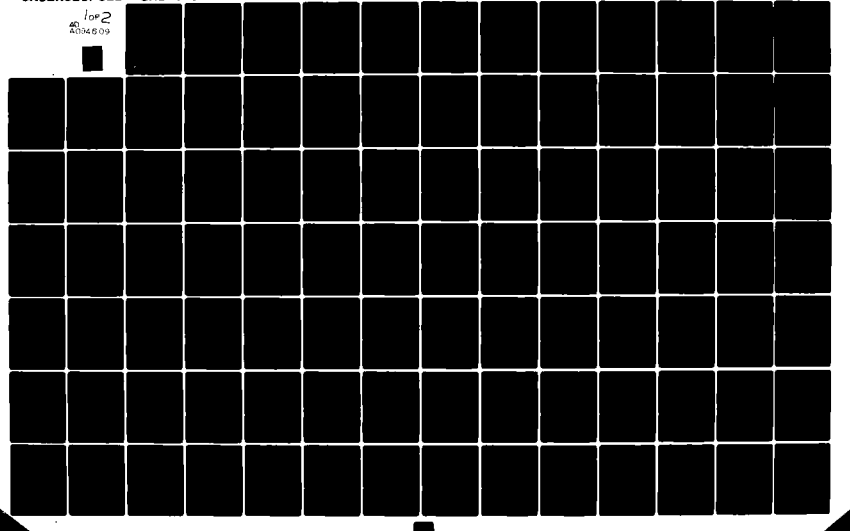
AD-A094 609 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 12/1
A VERIFICATION CONDITION GENERATOR FOR FORTRAN.(U)
JUN 80 R S BOYER, J S MOORE N00014-75-C-0816
UNCLASSIFIED SRI/CSL-103 NL



LEVEL II

A VERIFICATION CONDITION GENERATOR FOR FORTRAN

Technical Report CSL-103
SRI Project 4079

Contract No. N00014-75-C-0816

June 1980

By: Robert S. Boyer
J Strother Moore

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Office of Naval Research
Department of the Navy
Arlington, Virginia 22217

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-1246

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CSL-103	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A Verification Condition Generator for FORTRAN	5. TYPE OF REPORT & PERIOD COVERED: Technical
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Robert S. Boyer; J Strother Moore	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0816
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Laboratory, SRI International, Menlo Park, CA 94025	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 049-378
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Department of the Navy, Arlington, VA 22217	12. REPORT DATE: June 1980; 13. NO. OF PAGES: iv+134
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Reproduction in whole or in part is permittted for any purpose of the United States Government.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

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

18. SUPPLEMENTARY NOTES

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

proof of program properties
inductive invariants
automatic theorem-proving
string searching
aliasing
program semantics

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

This paper provides both a precise specification of a subset of FORTRAN 66 and FORTRAN 77 and a specification of the verification condition generator we have implemented for that subset. Our subset includes all the statements in FORTRAN 66 except the following: READ, WRITE, REWIND, BACKSPACE, ENDFILE, FORMAT, EQUIVALENCE, DATA, and BLOCK DATA. We place some restrictions on the remaining statements; however, our subset includes certain uses of COMMON, adjustable array dimensions, function subprograms, subroutine subprograms with side effects, and computed and assigned GO TOs. Unusual features of our system include a syntax.

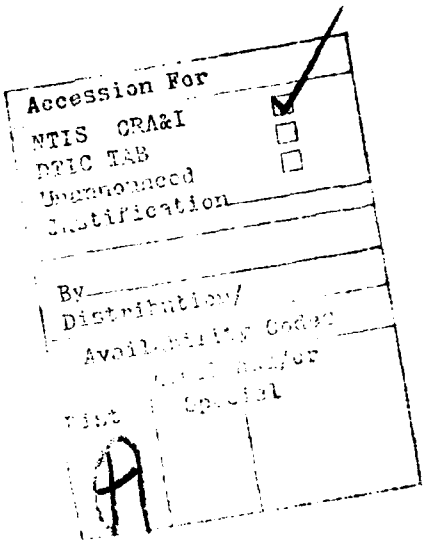
DD FORM 1 JAN 73 1473
EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

checker that enforces all our syntactic restrictions on the language, the thorough analysis of aliasing, the generation of verification conditions to prove termination, and the generation of verification conditions to ensure against such run-time errors as array bound violations and arithmetic overflow. We have used the system to verify several running FORTRAN programs. We present one such program and discuss its verification.



CONTENTS

I SUMMARY*

Mechanical program verification systems usually consist of two main programs -- a verification condition generator and a theorem-prover. The use of such a system involves two steps. (a) The verification condition generator accepts as its input a source program to be verified, an input/output specification for the program, and some inductive invariants; the verification condition generator produces some formulas, called "verification conditions," which imply that the source program behaves as specified. (b) The verification conditions are then submitted to the theorem-prover. If the theorem-prover determines that the verification conditions are theorems, then the source program behaves as specified.

This document describes a verification condition generator for a subset both of FORTRAN 66 [12] and FORTRAN 77 [1]. While we place constraints on the language that are not found in the ANSI specifications, ours is a true subset in the sense that a processor that correctly implements either FORTRAN correctly implements our language. Our subset includes certain uses of COMMON, function subprograms, subroutine subprograms with side effects, and computed and assigned GO TOs. The most notable exceptions from our subset are the input/output statements (e.g., READ, WRITE, and FORMAT), EQUIVALENCE, DATA, and procedural parameters.

The logical language used to specify the FORTRAN programs is that described in [5] and in [6]. The verification conditions produced are suitable for input to the theorem-prover described in [5].

Unusual features of our system -- aside from our choice of FORTRAN and our use of a quantifier free specification language -- include a

---

* The work reported here was supported in part by ONR Contract N00014-75-C-0816.

syntax checker that enforces all our syntactic restrictions on the language, the thorough analysis of aliasing, the generation of verification conditions to prove termination, and the generation of verification conditions to ensure against such run-time errors as array-bound violations and arithmetic overflow.

Although our syntax checker and verification condition generator handle programs involving finite precision REAL arithmetic, we have not yet formalized the semantics of those operations and hence cannot mechanically verify programs that operate on REALs.

The two step approach to program verification was formalized by Floyd in [8]. King [10] implemented the first mechanical verification condition generator. Since then, many verification condition generators have been implemented for many different programming languages, although we are not aware of any other verification condition generator for FORTRAN. For an introduction to program verification, see Anderson [2] or Manna [11]; both books contain bibliographies.

II APOLOGIA

The two steps of program verification stem from the fact that conventional (i.e., von Neumann) programming languages are not mathematical languages. The semantics of von Neumann languages are sufficiently messy that it is not possible to derive one truth from another in these languages by the application of simple rules such as "modus ponens" and "substitution of equals for equals." While various methods have been proposed for conducting "proofs" in these von Neumann languages, the methods are all variations of the two-step theme: transform the specified and annotated program into mathematical formulas and prove those formulas.

There exist programming languages that are also mathematical languages. In fact, the last few years have seen so much "flexibility" introduced into the notion of "programming language" that prominent researchers have agreed to call first order predicate calculus and set theory "programming languages" and then proceeded to argue the merits of various "compilers" and "interpreters" (i.e., theorem-provers). No one has yet found a way to execute programs written in such languages as efficiently as programs written in conventional programming languages -- at least for most of the usual programming tasks.

However, because of the great effort devoted to programming language design and semantics during the past decade, many students of program semantics will smirk, scowl, or choke at the mere mention of the word "FORTRAN." For example, a noted program semanticist remarked to the First International Conference on Reliable Software that if the West hoped to win the next war, it had better stop using COBOL and FORTRAN.*

---

* We leave to the reader the tough choice between (a) the simple semantics of almost-as-efficient modern languages providing such features as variant records and pointers, and (b) the blinding speed of a resolution theorem-prover executing the powerful "there exists x such that p(x)" feature.

Nevertheless, the use of FORTRAN is widespread, even within first-rate computer science departments. We suspect that the wide use of FORTRAN will continue until someone designs and implements a mathematical programming language that executes as efficiently as FORTRAN. As long as the use of FORTRAN continues, we suspect that it may be profitable to specify and verify FORTRAN programs.

We conclude this apologia with a quotation from Backus's "The History of FORTRAN, I, II, and III" [3].

> To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed. In fact, I believe that we are in a similar, but unrecognized, situation today: in spite of all the fuss that has been made over myriad language details, current conventional languages are still very weak programming aids, and far more powerful languages would be in use today if anyone had found a way to make them run with adequate efficiency. In other words, the next revolution in programming will take place only when both of the following requirements have been met: (a) a new kind of programming language, far more powerful than those of today, has been developed and (b) a technique has been found for executing its programs at not much greater cost than that of today's programs.

III THE ANSI SPECIFICATIONS OF FORTRAN

Neither specification of FORTRAN ([12], [1]) provides all that is needed to specify a verification condition generator.

(1) The rules that define the syntax of FORTRAN are clearly stated, but they are intermixed with those that define the execution of a FORTRAN program. Because the specification of the syntax and semantics are intertwined, the reader of this document might have some difficulty interpreting a remark by us such as "suppose we are given a syntactically correct program." Therefore, we specify in detail the syntax of our subset of FORTRAN.

(2) The results of arithmetic operations on types REAL, DOUBLE PRECISION, and COMPLEX are not specified in [12] or [1] (or, for that matter, in more fashionable language definitions). The absence of such specifications makes it difficult to verify much about programs that use such operations. It may be possible to specify such operations in a way that is applicable to a variety of existing machines and useful for mechanical theorem-proving; see, for example, Brown [7]. We have not yet incorporated any such specifications into our system. We repeat an early warning of Goldstine and von Neumann [9] on just this issue:

> The floating binary point represents an effort to render a thorough mathematical understanding of at least part of the problem unnecessary, and we feel that this is a step in a doubtful direction.

(3) The definitions of FORTRAN do not provide a method for specifying in a formal language the effects or results of FORTRAN subprograms. (For example, there is no specified nomenclature for referring to entities of a COMMON block not declared in that program unit even though the program unit might be specified to redefine those entities via CALLs to other subprograms.) We have invented some nomenclature for specifying input/output assertions, invariants, and so forth.

(4) Some concepts used in [12] and [1] (e.g., "entity," "by

value," and "by name") are not defined and some statements are bafflingly vague (e.g., "It is not necessary for a processor to evaluate all of the operands of an expression if the value of the expression can be determined otherwise." Section 6.6.1 [1]). We believe it is possible to produce paraphrases of [12] and [1] that are formal and that reflect the intentions of the authors, but no such documents exist as far as we know. We have relied upon common sense and our understanding of informal English to imagine a formal definition of FORTRAN with respect to which our verification condition generator might be formally proved correct.

IV AN INFORMAL DESCRIPTION OF OUR FORTRAN SUBSET

In this section we describe informally the subset of FORTRAN with which our system deals.

In selecting our subset, we omitted many features. That we have omitted a feature of FORTRAN does not indicate that we think that feature is logically intractable. We have no doubt that a verification condition generator could be implemented to include some of the FORTRAN features we have omitted.

For the rest of this section and the next, we assume that the reader has a rough idea of FORTRAN syntax.

The input to our verification condition generator must include not only the subprogram (function or subroutine) to be verified, but also all subprograms referenced somehow by the candidate subprogram. Each referenced subprogram must have been previously specified and verified. For example, some of our restrictions depend upon the types and dimensions of the "dummy arguments" (i.e., formal parameters) of the referenced subprograms and upon how those subprograms modify their arguments.

A. Statements in Our Subset

The FORTRAN statements in our subset are:

Arithmetic assignment	DO
Logical assignment	DIMENSION
GO TO assignment	COMMON
Unconditional GO TO	INTEGER
Assigned GO TO	REAL
Computed GO TO	DOUBLE PRECISION
Arithmetic IF	COMPLEX
CALL	LOGICAL
RETURN	EXTERNAL
CONTINUE	Statement function
STOP	FUNCTION
PAUSE	SUBROUTINE
Logical IF	END

Our subset does not include the following FORTRAN 77 statements:

BACKSPACE	FORMAT
BLOCK DATA	IMPLICIT
Block IF	INQUIRE
CHARACTER	INTRINSIC
Character assignment	OPEN
CLOSE	PARAMETER
DATA	PRINT
ELSE	PROGRAM
ELSEIF	READ
ENDFILE	REWIND
ENDIF	SAVE
ENTRY	WRITE
EQUIVALENCE	

B. Restrictions

For those statements in our subset we enforce all of the restrictions of both FORTRAN 66 and 77; furthermore, we enforce some additional restrictions.

To state our restrictions precisely we introduce some nomenclature. We do so formally later. One such notion is that a variable or array is "possibly smashed" by a subprogram. Roughly speaking, this means that the subprogram contains an assignment statement which alters the variable or array, or the subprogram calls another subprogram that possibly smashes the variable or array.

We now informally enumerate the major restrictions we impose, beyond those imposed by the ANSI specifications. While some of the restrictions may appear radical, many of the most severe are in fact closely related to ANSI restrictions. In the next section we comment on the relations between our restrictions and those of ANSI.

Every expression using infix operators must be fully parenthesized. For example, either (A + (B + C)) or ((A + B) + C) is permitted, but A + B + C is not.

We countenance no implicit coercion. In an arithmetic assignment statement or a statement function statement, v = e, the type of e must be the type of v. In $(e_1 + e_2)$ the types of e_1 and e_2 must be the same.

No Hollerith constants are permitted.

No COMMON statement may declare a variable or array to be in blank COMMON, and the components of each labeled COMMON block x must be specified in exactly the same order and with exactly the same names, types, and dimensions in each subprogram in which x is a labeled COMMON block.

The names of intrinsic functions cannot be used except to denote those functions. For example, ABS may not be used as the name of a user-defined function subprogram.

No variable used in a GO TO assignment or an assigned GO TO statement of a subprogram may be used in any statement of the subprogram except a type, assigned GO TO, or GO TO assignment statement.

Subroutines and functions may not be passed as arguments to subprograms.

In a CALL statement or function reference, if the formal argument is an array, then the corresponding actual must be an array of the same size and number of dimensions.

Function subprograms may not possibly smash any of their arguments or anything in COMMON. That is, function subprograms may not have side effects.

No subroutine call may "possibly" violate the strict aliasing restrictions of FORTRAN. For example, if a subroutine has two arguments and possibly smashes the first, then that subroutine may not be called with the same array passed in both arguments nor may an array in COMMON be passed as the first argument if the subroutine "knows" about the COMMON block, even via subprograms. Furthermore, an array element may not be passed to a subroutine in an argument position that is possibly smashed.

An adjustable array dimension may not be possibly smashed, and the control variable and parameters of a DO may not be possibly smashed within the range of the DO.

DOs may not have extended ranges.

C. Tokens

Suppose we have written and verified a subprogram in which a local array is declared to be of size 256. Suppose that we later wish to use the subprogram in another application and wish the local array to be of size 128. Then, if we wish to have confidence in the correctness of the modified program, we must verify it "again." For example, the new program may not have enough space to perform as specified, array bounds may be violated (either positively or negatively), and a new analysis of overflow and underflow is necessary.

Since twiddling the built-in constants in a program is a fairly common activity, especially when moving the program from one site to another, it is convenient if it can be done without incurring the cost of verifying the modified program. To that end we permit the simultaneous verification of a large class of programs by the addition to our language of what we call "tokens." From the programmer's point of view, tokens are similar to INTEGER variables, except that they may be used wherever FORTRAN permits INTEGER constants. Furthermore, before the subprogram is compiled, the user must specify positive INTEGER constants to be substituted for the tokens. Such a substitution into a syntactically correct program (as we define it) produces a syntactically correct FORTRAN program.

To prove the correctness of subprograms containing tokens, it is often necessary to include hypotheses about the values of the tokens in one's input assertions. For example, in one program we have verified we required that one token be a power of two and another be its base two logarithm. Not only do tokens make it easier to obtain two slightly different versions of a correct program, they usually make it easier to verify a single program because they make obvious the key relationships

between the constants without bringing in unnecessary detail (such as 8192 and 13). In addition, the explicit statement of the crucial relationships between the constants makes it easier to modify the program in the future. Of course, failure to instantiate the tokens with values satisfying the input assertions will produce programs that execute correctly whenever unsatisfiable input assertions are satisfied.

V COMMENTS ON THE RESTRICTIONS

In this section we offer partial explanations for the major restrictions enumerated in the previous section.

A. Full Parenthesization

FORTRAN permits one to write A+B+C. The order in which the subexpressions of unparenthesized expressions are combined is left up to the processor. However, (A+B)+C may cause an overflow when A+(B+C) does not (e.g., let A and B be very large INTEGERs and let C be the negation of B). By insisting upon full parenthesization, we reduce the number of orders of combination. However, even if an expression is fully parenthesized, a processor may use the facts that addition and multiplication are commutative.

B. Coercion

FORTRAN permits one to write R+D, where R is of type REAL and D is of type DOUBLE PRECISION. The result is of type DOUBLE PRECISION. Similarly, one may write the assignment D = R, which converts the value of R into a DOUBLE PRECISION number and smashes the result into D. For simplicity, we prohibit such implicit coercion. Since our subset includes the intrinsic functions for explicit coercion (e.g., DBLE converts a REAL value into a DOUBLE PRECISION one) no expressive power is lost (e.g., we permit DBLE(R)+D and D = DBLE(R)).

C. Hollerith Data

FORTRAN 66 makes some provisions for manipulating "Hollerith data." However, FORTRAN 77 does not.

D. COMMON

FORTRAN permits two subprograms to declare different organizations for the same COMMON block. Thus, one subprogram may declare that the first "storage unit" contains a REAL while the other subprogram declares that the first storage unit contains an INTEGER. The two declarations need not have the same number of names, sizes, or types. For labeled COMMON they must, however, describe the same number of storage units. For blank COMMON, one may be arbitrarily longer than the other.

Such organization of storage complicates the determination of whether a variable is defined and what its value is. For example, if the INTEGER variable I and the REAL variable R share the same storage location, then I becomes undefined when R is defined (e.g., assigned to), and vice versa. We made our restrictions on the use of COMMON to eliminate such complications.

E. Intrinsic Functions

FORTRAN permits the user to define a function with the same name as an intrinsic function. But FORTRAN 66 and 77 have different rules for obtaining the right to call such user-defined functions.

F. Label Variables

We enforce a strict syntactic segregation between INTEGER variables used to hold statement labels and INTEGER variables used in normal arithmetic expressions.

FORTRAN achieves much the same effect but makes a semantic requirement. In particular, if an INTEGER variable is assigned a label, subsequent reference to the variable as an INTEGER is prohibited unless there is an intervening arithmetic assignment. Similarly, an assigned GO TO is prohibited if the label variable has an arithmetic rather than label value. Our restriction does not decrease the expressive power of the language, but may require the introduction of a second variable name to be used in those contexts requiring a "statement label type" variable.

G. Functional Parameters

Our verification condition generator assumes that each time a CALL statement or function reference is encountered it can determine exactly which subroutine or function subprogram is being invoked so that it can obtain the specification of the referenced subprogram and produce the necessary verification conditions. If we had permitted subroutines or functions to be passed as arguments, then verification condition generation would have been severely complicated.

H. Agreement of Dimension

FORTRAN permits the passing of an n-dimensional array to a subprogram "expecting" an m-dimensional array. Since the ANSI specifications spell out the order in which array elements are stored, such abuse of array indexing is well defined and can be used to implement unusual overlays and access patterns.

FORTRAN also permits passing an array element to a subprogram expecting an array. We have omitted both features because of their complexity.

I. Functions

Our requirement that function subprograms not have side effects stems from the fact a FORTRAN processor may evaluate (or not evaluate) the parts of an expression in an unpredictable order.

Let us illustrate the problems this causes. Suppose that N is an INTEGER variable in COMMON, that function subprogram R has the side effect of setting N to N+1 and that function subprogram S has the side effect of setting N to 2*N. Finally, suppose that just before executing the assignment statement

```
X = (R(X)*S(X))
```

N has the value 5. Then after the execution of the assignment statement, N may have the value 11 or may have the value 12.

Worse, if the value returned by S(X) is 0, then the value of N after the evaluation of (R(X)*S(X)) would be 10 if the processor were smart enough not to evaluate R(X) once it spotted that S(X) returned 0, since an expression part need not be evaluated unless the processor finds it necessary. In fact, a really smart processor might be able to determine that S(X) always returns 0. Then after the evaluation of (R(X)*S(X)), the value of N might still be 5!

Of course, the reason FORTRAN permits such flexibility in evaluation is so that good optimizing compilers can be written. In fact, the ANSI definitions of FORTRAN specify that the value of N is "undefined" if either the call to R or S might not be evaluated. (See section 10.2.9 of [12] and section 6.6.1 of [1]). It is interesting to speculate on the number of times correct optimizing compilers have introduced "bugs" into programs written by programmers unfamiliar with the ANSI specifications.

Another reason for prohibiting side effects in function calls is the complexity of the rules concerning what is perhaps the most bizarre concept in [12] -- "second level definition." (See sections 10.2.7 and 10.2.8.) It is perhaps not widely known that if X is of type REAL, A is an array, and R and N are as above, then the following sequence of two instructions is illegal:

```
X=R(X)
B=A(N)
```

After the first statement, in which N is redefined, N is no longer defined at the second level and may not be used as a subscript.

J. Aliasing

Presumably because of a desire to permit the implementation of parameter passing by either a "call by reference" or "call by return value" scheme, and because of its concern with optimizing compilers, the FORTRAN 66 specification took a dim view of aliasing.

For example, suppose subroutine SUBR takes two arguments, I and J, and assigns to I. Then a CALL of the form

```
CALL SUBR(K, K)
```

is illegal. In a "call by reference" implementation, the assignment to I would also smash J inside the subprogram. (Consider the difficulty of writing an optimizing compiler for such an implementation of parameter passing unless the specification rules out CALLs such as the one exhibited. For example, it would be impossible to take advantage of the information that the current value of J is in an accumulator.) In a "call by return value" implementation the assignment to I would not smash J, but the final effect of the CALL on K would depend upon the order in which the final values of the formal parameters were assigned to K.

Without mentioning the word "aliasing," FORTRAN 66 prohibits it in some cases. "If a subroutine reference causes a dummy argument [i.e., formal parameter] in the referenced subroutine to become associated with [i.e., be allocated the same storage location as] another dummy argument in the same subroutine or with an entity in COMMON, a definition of [i.e., assignment to] either entity within the subroutine is prohibited." (Section 8.4.2 of [12], bracketed definitions of unconventional terms added by us). FORTRAN 77 has a similar prohibition.

We enforce this restriction by requiring that if a subprogram "possibly smashes" an argument, then it is illegal to associate that argument with any other argument or entity in COMMON. (A less syntactic interpretation is possible. We consider illegal CALLs that "possibly smash" aliased arguments even if the arguments are not "actually" smashed.)

K. DO Loop Controls

FORTRAN 66 specifies that variables that represent adjustable array dimensions in a subprogram may not be modified during execution of the subprogram. In addition the control variable and parameters of a DO statement may not be modified by the statements in the range of the DO. Oddly, FORTRAN 77 permits both kinds of modifications but specifies that such modifications do not affect the size of the array or the number of times the DO loop is executed. This permits compilers to optimize the control of DO loops by storing the initial values of those variables in registers and using hardware increment and test instructions.

Again, we use our syntactically defined concept of "possibly smashed" to interpret this. For example, if SUBR is a subroutine that possibly smashes its first argument and I is the control variable of a DO statement, then the range of the DO statement may not contain:

```
CALL SUBR(I, X)
```

because it appears to modify I illegally. It is possible that particular invocation of SUBR does not in fact modify I so that such a CALL is technically permitted in FORTRAN.

L. Extended DOs

Although FORTRAN 66 permits "extended DOs," FORTRAN 77 prohibits them.

VI THE FORMAL SYNTAX

In this and the next five sections we make precise the preceding vague statements about our FORTRAN subset and what we mean when we say a subprogram is "correct" with respect to an input/output specification. Then, in Section XIII, we present a sample program in our subset and some of the verification conditions generated for it. The example has been organized so as to illustrate some of the formal notions about to be presented. However, some readers may wish to inspect the example before entering the formal sections of this document.

Lasciate ogni speranza, voi ch' entrate.

We adopt from [1] the definitions of the following: symbolic name, INTEGER constant, REAL constant, DOUBLE PRECISION constant, COMPLEX constant, LOGICAL constant, and unsigned constant.

For example, MATRIX is a symbolic name. -127 is an INTEGER constant and 1.25E-5 is a REAL constant.

Notation. sequences. We shall write

<> for the empty sequence,

<a> for the sequence of length 1 whose only member is a,

<a b> for the sequence of length 2 whose first member is a and whose second member is b,

<a b c> for the sequence of length 3 whose first member is a, whose second member is b, and whose third member is c,

and so on.

Definition. token. A token is a sequence of 2 to 7 characters that satisfies the constraints on variable symbols in our logic (see [5] and [6]) and that begins with the character @.

Definition. constant. x is a constant if and only if x is an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL constant.

Definition. label. A label is a sequence of from 1 to 5 digits the first of which is not 0.

Definition. types. x is a type if and only if x is one of the character sequences INTEGER, DOUBLE, COMPLEX, REAL, or LOGICAL.

Definition. type of a constant. The type of an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL constant is respectively INTEGER, REAL, DOUBLE, COMPLEX, or LOGICAL.

Definition. variable pattern. x is a variable pattern if and only if for some symbolic name n and some type t, x is <n t>. If <n t> is a variable pattern, then n is its name and t is its type.

Note. We have invented the concept of variable pattern, which is not employed in the usual specifications of FORTRAN syntax, to help in our formulation of FORTRAN syntax (e.g., in the syntax of well-formed FORTRAN expressions). The presence of the variable pattern <v t> in the "syntactic environment" (to be defined) of a FORTRAN program indicates that the symbolic name v is to be used as a variable of type t.

Definition. array pattern. x is an array pattern on s if and only if s is a set of symbolic names and for some n, t, i, j, and k, each of the following is true:

(1) x is one of the sequences <n t i>, <n t i j>, or <n t i j k>.

(2) n is a symbolic name.

(3) t is a type.

(4) Each of i, j, and k is either a positive, unsigned INTEGER constant, a token, or a member of s.

The name of an array pattern is its first member, its type is its second member. The dimension list of an array pattern p is the terminal subsequence of p starting with the third member of p.

Note. An array pattern is used to encode the type, number of dimensions, and size of each dimension of a symbolic name used as an

array name in a FORTRAN subprogram. Those elements of the dimension list that are symbolic names are the so-called "adjustable" dimensions of the array. For example, if JMAX is in the set s, then <MATRIX REAL 10 JMAX> is an array pattern on s and encodes the information that MATRIX is a two-dimensional array of type REAL, measuring 10 by JMAX.

Definition. sort. s is a sort if and only if for some t, i, j, and k, t is a type, each of i, j, and k is a positive INTEGER, a token, or a symbolic name, and s is one of the sequences

<t>
<t i>
<t i j>
<t i j k>.

The sort of a variable pattern <v t> is <t>, the sort of an array pattern <v t i>, <v t i j>, or <v t i j k> is, respectively, <t i>, <t i j>, or <t i j k>.

Definition. function pattern. p is a function pattern if and only if for some fn, some t, and some nonempty sequence $\langle v_1 \ldots v_n \rangle$ each of the following statements is true:

(1) p is the sequence $\langle$fn t $v_1 \ldots v_n \rangle$.

(2) fn is a symbolic name.

(3) t is a type.

(4) Each member of $\langle v_1 \ldots v_n \rangle$ is either a variable pattern or an array pattern on the set of names of the v_i with type INTEGER.

(5) fn is not the name of any member of $\langle v_1 \ldots v_n \rangle$.

(6) For each choice of v_i and v_j from $\langle v_1 \ldots v_n \rangle$, the name of v_i is different from that of v_j, provided i and j are different.

The name of a function pattern is its first member. The type of a function pattern is its second member. If p is a function pattern of length n+2, then (a) p has n arguments, (b) if i is an integer greater than 0 and less than or equal to n, then the ith argument of p is the i+2nd member of p, and (c) the argument list of p is the sequence of length n whose ith member is the ith argument of p.

Note. A function pattern is used to encode the number, names, and sorts of dummy arguments to a function subprogram and the type of the result.

Definition. statement function pattern. p is a statement function pattern if and only if p is a function pattern and each member of the argument list of p is a variable pattern.

Note. A statement function pattern is used to encode the number and types of the arguments of a statement function and the type of the result. The names of the variable patterns in a statement function pattern are actually irrelevant. See the definition of the "statement function statement."

Definition. intrinsic function pattern. The intrinsic function patterns are:

```
<ABS REAL <I REAL>>
<IABS INTEGER <I INTEGER>>
<DABS DOUBLE <I DOUBLE>>
<AINT REAL <I REAL>>
<INT INTEGER <I REAL>>
<IDINT INTEGER <I DOUBLE>>
<AMOD REAL <I REAL> <J REAL>>
<MOD INTEGER <I INTEGER> <J INTEGER>>
<AMAX0 REAL <I INTEGER> <J INTEGER>>
<AMAX1 REAL <I REAL> <J REAL>>
<MAX0 INTEGER <I INTEGER> <J INTEGER>>
<MAX1 INTEGER <I REAL> <J REAL>>
<DMAX1 DOUBLE <I DOUBLE> <J DOUBLE>>
<AMIN0 REAL <I INTEGER> <J INTEGER>>
<AMIN1 REAL <I REAL> <J REAL>>
<MIN0 INTEGER <I INTEGER> <J INTEGER>>
<MIN1 INTEGER <I REAL> <J REAL>>
<DMIN1 DOUBLE <I DOUBLE> <J DOUBLE>>
<FLOAT REAL <I INTEGER>>
<IFIX INTEGER <I REAL>>
<SIGN REAL <I REAL> <J REAL>>
<ISIGN INTEGER <I INTEGER> <J INTEGER>>
<DSIGN DOUBLE <I DOUBLE> <J DOUBLE>>
<DIM REAL <I REAL> <J REAL>>
<IDIM INTEGER <I INTEGER> <J INTEGER>>
<SNGL REAL <I DOUBLE>>
<REAL REAL <I COMPLEX>>
<AIMAG REAL <I COMPLEX>>
<DBLE DOUBLE <I REAL>>
<CMPLX COMPLEX <I REAL> <J REAL>>
```

```
<CONJG COMPLEX <I COMPLEX>>
<EXP REAL <I REAL>>
<DEXP DOUBLE <I DOUBLE>>
<CEXP COMPLEX <I COMPLEX>>
<ALOG REAL <I REAL>>
<DLOG DOUBLE <I DOUBLE>>
<CLOG COMPLEX <I COMPLEX>>
<ALOG10 REAL <I REAL>>
<DLOG10 DOUBLE <I DOUBLE>>
<SIN REAL <I REAL>>
<DSIN DOUBLE <I DOUBLE>>
<CSIN COMPLEX <I COMPLEX>>
<COS REAL <I REAL>>
<DCOS DOUBLE <I DOUBLE>>
<CCOS COMPLEX <I COMPLEX>>
<TANH REAL <I REAL>>
<SQRT REAL <I REAL>>
<DSQRT DOUBLE <I DOUBLE>>
<CSQRT COMPLEX <I COMPLEX>>
<ATAN REAL <I REAL>>
<DATAN DOUBLE <I DOUBLE>>
<ATAN2 REAL <I REAL> <J REAL>>
<DATAN2 DOUBLE <I DOUBLE> <J DOUBLE>>
<DMOD DOUBLE <I DOUBLE> <J DOUBLE>>
<CABS REAL <I COMPLEX>>
```

Note. In the spirit of FORTRAN 77, our intrinsic function patterns include patterns for the basic external functions of FORTRAN 66.

Our FORTRAN subset includes operations on types REAL, DOUBLE, and COMPLEX, and our verification condition generator (to be described) will generate correct verification conditions for those operations provided their input/output relations are specified. However, we have not yet specified any of the operations involving finite precision REAL arithmetic and consequently have no mechanical means of proving anything about FORTRAN programs that use such operations.

Our patterns for the maximum and minimum functions have only two arguments. Consequently, syntactically correct programs in our subset may not apply the maximum and minimum functions to an arbitrary number of arguments as permitted in FORTRAN. The maximum and minimum functions are the only FORTRAN 66 functions with an indefinite number of arguments.

Definition. subroutine pattern. p is a subroutine pattern if and only if for some fn and some (possibly empty) sequence $v_1, \ldots, v_n$ each of the following statements is true:

(1) p is the sequence $\langle fn\ v_1 \ldots v_n \rangle$.

(2) fn is a symbolic name.

(3) Each member of $\langle v_1 \ldots v_n \rangle$ is either a variable pattern or an array pattern on the set of names of the v_i with type INTEGER.

(4) fn is not the name of any member of $\langle v_1 \ldots v_n \rangle$.

(5) For each possible choice of v_i and v_j from $\langle v_1 \ldots v_n \rangle$, the name of v_i is different from that of v_j provided i is different from j.

The name of a subroutine pattern is its first member. If p is a subroutine pattern of length n+1, then (a) p has n arguments, (b) if i is an integer greater than 0 and less than n+1, then the ith argument of p is the i+1st member of p, and (c) the argument list of p is the sequence of length n whose ith member is the ith argument of p.

Note. We next define the notion of a "syntactic environment." Intuitively, such an environment is implicitly associated with a given FORTRAN subprogram (e.g., subroutine) and specifies the names of all entities known to the subprogram: arrays, variables (artificially divided into two sets according to whether they will be used to store labels for assigned GO TO statements), functions (divided into statement functions and others), subroutines, and COMMON block names. A syntactic environment also specifies other syntactic information about these entities, such as their type, number of dimensions or arguments, and so on.

Definition. syntactic environment. s is a syntactic environment if and only if s is a sequence of seven sets (called the array patterns, variable patterns, label variable patterns, statement function patterns, function patterns, subroutines patterns, and block names of s) such that each of the following statements is true:

(1) Each member of the array patterns is an array pattern on the names of the variable patterns of type INTEGER, and no two members of the array patterns have the same name.

(2) Each member of the variable patterns is a variable pattern, and no two members of the variable patterns have the same name.

(3) Each member of the label variable patterns is a variable pattern of type INTEGER.

(4) Each member of the statement function patterns is a statement function pattern, and no two members of the statement function patterns have the same name.

(5) Each member of the function patterns is a function pattern, and no two members of the function patterns have the same name.

(6) Each member of the subroutine patterns is a subroutine pattern, and no two members of the subroutine patterns have the same name.

(7) Each member of block names is a symbolic name.

(8) If n is the name of a member of one of the first six sets (i.e., the array, variable, label variable, statement function, function, or subroutine patterns of s), then n is not the name of a member of any other member of the first six sets nor is n a member of the block names.

(9) If n is a member of the block names of s, then it is not the name of any member of any of the first six sets.

(10) If n is the name of an intrinsic function pattern, then n is not a member of the block names of s nor is it the name of any member of the array, variable, label variable, statement function, or subroutine patterns of s.

(11) If n is the name of an intrinsic function pattern and n is the name of a member of the function patterns of s, then the intrinsic function pattern of which n is the name is a member of the function patterns of s.

Note. Most of the restrictions above regarding the use of names of various types follow from the ANSI FORTRAN specifications. Here are the major additional requirements we impose:

(1) the strict segregation of "normal" variables from those variables involved in assigned GO TO or GO TO assignment statements,

(2) the limitation of the use of intrinsic function names, and

(3) the disjointness requirements on the names in the seven sets above. (FORTRAN permits a limited amount of overlapping, e.g., the variable names must be disjoint from the array names, but not necessarily from the block names.)

Notation. has the form. x has the form y if and only if x and y are sequences of characters and x is the result of replacing each maximal, contiguous subsequence of y composed of lower case alphanumeric characters with the current meaning of each of those subsequences. For example, if a is "XY", ab is "UV", and z is "ABC", then "IF UV(XY) = ABC" has the form

IF ab(a) = z

Definition. subscript. x is a subscript with respect to s if and only if s is a syntactic environment and x is an unsigned INTEGER constant, a token, or the name of some variable pattern of type INTEGER in the variable patterns of s.

Definition. expression. Suppose s is a syntactic environment. We define inductively the concept c is an expression with respect to s. Simultaneously, we define the sort of c with respect to s and we define the proper subexpressions of c. We shall omit the phrase "with respect to" through this definition and wherever the appropriate s is obvious *from context.*

(1) If c is an unsigned constant of type t, then c is an expression, <t> is the sort of c, and c has no proper subexpressions.

(2) If c is a token, then c is an expression, <INTEGER> is the sort of c, and c has no proper subexpressions.

(3) If c is the name of a member of the variable patterns of s, then c is an expression, the sort of c is the sort of the variable pattern with name c in the variable patterns of s, and c has no proper subexpressions. [Such a c is called a variable reference with respect to s.]

(4) If c is the name of a member of the array patterns of s, then c is an expression, the sort of c is the sort of the array pattern with name c in the array patterns of s, and c has no proper subexpressions. [Such a c is called an array reference with respect to s.]

(5) For all a, t, i, j, and k, and for all subscripts e_1, e_2, and e_3,

   (a) if <a t i> is a member of the array patterns of s and c has the form

$a(e_1)$

then c is an expression, the sort of c is <t>, and the proper subexpressions of c are e_1 and the proper subexpressions of e_1;

(b) if <a t i j> is a member of the array patterns of s and c has the form

$a(e_1, e_2)$

then c is an expression, the sort of c is <t>, and the proper subexpressions of c are e_1, e_2, the proper subexpressions of e_1, and the proper subexpressions of e_2; and

(c) if <a t i j k> is a member of the array patterns of s and c has the form

$a(e_1, e_2, e_3)$

then c is an expression, the sort of c is <t>, and the proper subexpressions of c are e_1, e_2, e_3, the proper subexpressions of e_1, the proper subexpressions of e_2, and the proper subexpressions of e_3.

[Such a c is called an array element reference to a with respect to s. The subscript sequence of c is $\langle e_1 \rangle$, $\langle e_1\ e_2 \rangle$, or $\langle e_1\ e_2\ e_3 \rangle$, according to whether case (a), (b), or (c) above obtains.]

(6) For all e_1, e_2, and t, if e_1 and e_2 are expressions, t is a type other than LOGICAL, the sort of both e_1 and e_2 is <t>, and c has one of the forms

$(e_1 + e_2)$
$(e_1 - e_2)$
$(e_1 * e_2)$
(e_1 / e_2)

then c is an expression, the sort of c is <t>, and the proper subexpressions of c are e_1, e_2, the proper subexpressions of e_1, and the proper subexpressions of e_2. [Such a c is called an arithmetic expression with respect to s, the operation symbol of c is +, -, *, or /, according to which of the above four forms describes c, and the argument sequence of c is $\langle e_1\ e_2 \rangle$.]

(7) For all e_1, e_2, t_1, t_2 and t_3, if e_1 and e_2 are expressions, the sort of e_1 is $\langle t_1 \rangle$, the sort of e_2 is $\langle t_2 \rangle$, the sequence $\langle t_1\ t_2\ t_3 \rangle$ is one of the sequences:

<INTEGER INTEGER INTEGER>
<REAL INTEGER REAL>
<DOUBLE INTEGER DOUBLE>
<COMPLEX INTEGER COMPLEX>
<REAL REAL REAL>
<REAL DOUBLE DOUBLE>
<DOUBLE REAL DOUBLE>
<DOUBLE DOUBLE DOUBLE>

and c has the form

$(e_1$ ** $e_2)$

then c is an expression, the sort of c is $<t_3>$, and the proper subexpressions of c are e_1, e_2, the proper subexpressions of e_1, and the proper subexpressions of e_2. [Such a c is called an arithmetic expression with respect to s, the operation symbol of c is **, and the argument sequence of c is $<e_1\ e_2>$.]

(8) For all e_1, e_2, and t, if e_1 and e_2 are expressions, t is a type, t is not LOGICAL, t is not COMPLEX, both e_1 and e_2 have sort <t>, and c has one of the forms

$(e_1$ .LT. $e_2)$
$(e_1$ .LE. $e_2)$
$(e_1$ .EQ. $e_2)$
$(e_1$ .NE. $e_2)$
$(e_1$ .GT. $e_2)$
$(e_1$ .GE. $e_2)$

then c is an expression, the sort of c is <LOGICAL>, and the proper subexpressions of c are e_1, e_2, the proper subexpressions of e_1, and the proper subexpressions of e_2. [Such a c is called a relational expression with respect to s, the operation symbol of c is .LT., .LE., .EQ., .NE., .GT., or .GE. according to which of the above six forms describes c, and the argument sequence of c is $<e_1\ e_2>$.]

(9) For all e_1 and e_2, if e_1 and e_2 are expressions of sort <COMPLEX>, and c has the form

$(e_1$ .EQ. $e_2)$

or

$(e_1$ .NE. $e_2)$

then c is an expression, the sort of c is <LOGICAL>, and the proper subexpressions of c are e_1, e_2, the proper subexpressions of e_1, and the proper subexpressions of e_2. [Such a c is called a relational expression with respect to s, the operation symbol of c is .EQ. or .NE. according to which of the two forms above describes c, and the argument sequence of c is $\langle e_1\ e_2 \rangle$.]

(10) For all e_1 and e_2, if e_1 and e_2 are expressions and the sort of both e_1 and e_2 is <LOGICAL>, then if c has the form

$(e_1$ .OR. $e_2)$

or

$(e_1$ .AND. $e_2)$

then c is an expression, the sort of c is <LOGICAL>, and the proper subexpressions of c are e_1, e_2, the proper subexpressions of e_1, and the proper subexpressions of e_2. [Such a c is called a logical expression with respect to s, the operation symbol of c is .OR. or .AND. according to which of the above two forms describes c, and the argument sequence of c is $\langle e_1\ e_2 \rangle$.]

(11) For all e, if e is an expression, the sort of e is <LOGICAL>, and c has the form

(.NOT. e)

then c is an expression, the sort of c is <LOGICAL>, and the proper subexpressions of c are e and the proper subexpressions of e. [Such a c is called a logical expression with respect to s, the operation symbol of c is .NOT., and the argument sequence of c is <e>.]

(12) For each symbolic name fn, type t, integer n greater than 0, and for all sequences $\langle e_1 \ldots e_n \rangle$ and $\langle v_1 \ldots v_n \rangle$, if

(a) $\langle$fn t $v_1 \ldots v_n\rangle$ is a member of the statement function or the function patterns of s,

(b) for each integer i greater than 0 and less than or equal to n, e_i is an expression and

(i) the type of e_i is the type of v_i,

(ii) the length of the sort of e_i is the length of the sort of v_i, and

(iii) for k greater than 1 and less than or equal to the length of the sort of v_i, if the kth element of the sort of v_i is an INTEGER constant or token, then the kth element of the sort of e_i is that constant or token, and otherwise, for the m such that the name of v_m is the kth element of the sort of v_i, e_m is the kth element of the sort of e_i, and

(c) c has the form

$$fn(e_1, \ldots, e_n)$$

then c is an expression, the sort of c is <t>, and the proper subexpressions of c are the members of $<e_1 \ldots e_n>$ together with the proper subexpressions of each member of $<e_1 \ldots e_n>$. [Such a c is called a function reference to fn with respect to s, and the argument sequence of c is $<e_1 \ldots e_n>$.]

Note. In Section V we comment on some of the differences between our definition of an expression and the slightly more relaxed FORTRAN definitions (e.g., our prohibition of implicit coercion, our requirement of full parenthesization, and our requirement that arguments passed to functions have exactly the right dimension and size).

Definition. subexpressions. x is a member of the subexpressions of y if and only if for some syntactic environment s, x and y are expressions with respect to s and either x is y or x is a proper subexpression of y.

Definition. used as a subscript. x is used as a subscript in y with respect to s if and only if s is a syntactic environment, y is an expression with respect to s, and for some z, z is a subexpression of y, z is an array element reference with respect to s, and x is a symbolic name and a member of the subscript sequence of z.

Example. Suppose <A INTEGER 10> and <B INTEGER 10> are members of the array patterns of a syntactic environment s and that <I INTEGER> and <J INTEGER> are members of the variable patterns of s. Then I and J are the only expressions used as subscripts in ((A(I)+B(J))*A(3)).

Note. We now define the statements in our FORTRAN subset. Our subset includes the FORTRAN "DO statement," which is the standard iterative construct. However, because the semantics of the DO statement is naturally specified in terms of more primitive statements, we ignore the DO statement in our definition of syntactic and semantic correctness. After the definition of semantic correctness we describe how we handle the syntax and semantics of DO statements.

Definition. statement. st is a statement with respect to s if and only if s is a syntactic environment and one of the following statements is true:

(1) For some member n of the block names of s and some nonempty sequence $\langle a_1 \ldots a_k \rangle$ of distinct symbolic names, each member of which is the name of a member of the array or the variable patterns of s, st has the form

COMMON /n/a_1, a_2, ..., a_k

[Such a statement is called a COMMON statement. In such a statement, n is said to be declared as a COMMON block and each a_i is said to be declared to be in the COMMON block n.]

(2) For some v, t, i, j, and k, one of <v t i>, <v t i j>, or <v t i j k> is a member of the array patterns of s, and st has (respectively) the form

DIMENSION v(i)
DIMENSION v(i, j)
DIMENSION v(i, j, k)

[Such a statement is called a DIMENSION statement. In such a statement, v is declared to be an array.]

(3) For some v, v is the name of a variable, label variable, array, statement function, or function pattern of s whose type is INTEGER, REAL, DOUBLE, COMPLEX, or LOGICAL and st has (respectively) the form

INTEGER v
REAL v
DOUBLE PRECISION v
COMPLEX v
LOGICAL v

[Such a statement is said to be a type statement. In such a statement, v is declared to have type INTEGER, REAL, DOUBLE, COMPLEX, or LOGICAL, respectively.]

(4) For some v, v is the name of a nonintrinsic function pattern of s, and st has the form

EXTERNAL v

[Such a statement is said to declare v to be EXTERNAL.]

(5) For some t, v, and exp, t is a type, v and exp are both expressions of sort <t>, v is an array element reference or a variable reference, and st has the form

v = exp

[Such a statement is called an assignment statement.]

(6) For some i and k, i is the name of a label variable pattern of s, k is a label, and st has the form

ASSIGN k TO i

[Such a statement is called a GO TO assignment statement. k and only k is used as a label in such a statement.]

(7) For some label k, st has the form

GO TO k

[Such a statement is called an unconditional GO TO statement. k and only k is used as a label in such a statement.]

(8) For some label variable pattern of s with name i and for some nonempty sequence of labels $<k_1 \ldots k_n>$, st has the form

GO TO i, $(k_1, \ldots, k_n)$

[Such a statement is called an assigned GO TO statement. k is used as a label in such a statement if and only if k is a member of $<k_1 \ldots k_n>$.]

(9) For some variable pattern of s with name i and type INTEGER and for some nonempty sequence of labels $<k_1 \ldots k_n>$, st has the form

GO TO $(k_1, \ldots, k_n)$, i

[Such a statement is called a computed GO TO statement. k is used as a label in such a statement if and only if k is a member of $<k_1 \ldots k_n>$.]

(10) For some labels k_1, k_2, and k_3, and for some expression x of sort <INTEGER>, <REAL>, or <DOUBLE>, st has the form

IF (x) k_1, k_2, k_3

[Such a statement is called an arithmetic IF statement. k_1, k_2, and k_3, and only k_1, k_2, and k_3, are used as labels in such a statement.]

(11) For some subroutine pattern of s with name sub and (possibly empty) argument list $\langle v_1 \ldots v_n \rangle$, for some sequence $\langle a_1 \ldots a_n \rangle$ of expressions, and for all i greater than 0 and less than or equal to n, each of the following statements is true:

(a) the type of a_i is the type of v_i,

(b) the length of the sort of a_i is the length of the sort of v_i, and

(c) for k greater than 1 and less than or equal to the length of the sort of v_i, if the kth element of the sort of v_i is an INTEGER constant or token, then the kth element of the sort of a_i is that constant or token, and otherwise, for the m such that the name of v_m is the kth element of the sort of v_i, a_m is the kth element of the sort of a_i, and

(d) st has either the form

CALL sub

or the form

CALL sub(a_1, ..., a_n)

(according to whether n is 0 or greater than 0).

[Such a statement is a CALL of sub.]

(12) st has one of the forms

RETURN
CONTINUE
STOP
PAUSE

[Such a statement is called a RETURN, CONTINUE, STOP, or PAUSE statement, respectively.]

(13) For some sequence of digits, n, whose length is greater than 0 and less than 6, none of whose members is 8, and none of whose members is 9, st has the form

STOP n

or

PAUSE n

[Such a statement is called a STOP or PAUSE statement, respectively.]

(14) For some expression x of sort <LOGICAL> and for some st_2, st_2 is an assignment, GO TO assignment, unconditional GO TO, assigned GO TO, computed GO TO, arithmetic IF, CALL, RETURN, CONTINUE, STOP, or PAUSE statement with respect to s, and st has the form

IF (x) st_2

[Such a statement is called a logical IF statement and contains st_2. k is used as a label in such a statement if and only if st_2 is an unconditional, computed, or assigned GO TO, arithmetic IF, or GO TO assignment statement and k is used as a label in st_2.]

(15) For some member <fn t <v_1 t_1> ... <v_n t_n>> of the statement function patterns of s, for some nonempty sequence <<a_1 t_1> ... <a_n t_n>> of distinct members of the variable patterns of s, and for some expression x, not a variable reference, whose sort is <t> and which has no subexpression that is an array reference or an array element reference, st has the form

$f(a_1, \ldots, a_n) = x$

[Such a statement is called a statement function statement and is a definition of f. x is the body of such a statement and $a_1, \ldots, a_n$ are the arguments of such a statement.]

(16) For some variable pattern of s with name f and type t and for some nonempty sequence <a_1 ... a_n> of names of distinct members of the variable or array patterns of s, f is not one of the a_i, and either (a) t is not DOUBLE and st has the form

t FUNCTION $f(a_1, \ldots, a_n)$

or (b) t is DOUBLE and st has the form

DOUBLE PRECISION FUNCTION $f(a_1, \ldots, a_n)$

[Such a statement is called a FUNCTION statement. The arguments of the statement are $a_1, \ldots, a_n$. The name of the statement is f and the type of the statement is t.]

(17) For some symbolic name sub that is not a member of the block names of s and is not the name of any member of the array, variable, label variable, statement function, function, or subroutine patterns of s, and is not the name of any intrinsic function pattern, and for some (possibly empty) sequence $\langle a_1 \ldots a_n \rangle$ of names of distinct members of the variable and array patterns of s, st has the form

SUBROUTINE sub

or

SUBROUTINE sub(a_1, ..., a_n)

according to whether the sequence $\langle a_1 \ldots a_n \rangle$ is empty or not. [Such a statement is called a SUBROUTINE statement. sub is the name of the statement. In the first case, the statement has no arguments. In the second case the arguments of the statement are a_1, ..., a_n.]

(18) st has the form

END

[Such a statement is called an END statement.]

Note. In Section V we compare the syntax of statements in our subset with the syntax of FORTRAN statements.

Definition. executable statement. st is an executable statement with respect to s if and only if st is an assignment, GO TO assignment, unconditional GO TO, computed GO TO, assigned GO TO, arithmetic IF, CALL, RETURN, CONTINUE, STOP, PAUSE, or logical IF statement with respect to s.

Definition. used as a label variable. x is used as a label variable in st with respect to s if and only if s is a syntactic environment, x is a symbolic name, st is a statement with respect to s, and one of the following statements is true:

(1) For some labels lab_1, ..., lab_n, st has the form:

GO TO x, (lab_1, ..., lab_n)

(2) For some label lab, st has the form:

ASSIGN lab TO x

(3) For some statement st_2 with respect to s, st is a logical IF statement containing st_2 and x is used as a label variable in st_2.

Definition. used on the second level. x is used on the second level in st with respect to s if only if s is a syntactic environment, x is a symbolic name, st is a statement with respect to s, and one of the following statements is true:

(1) For some v and exp, st is an assignment statement of the form

v = exp

and x is used as a subscript in v or in exp.

(2) For some labels lab_1, ..., and lab_k, st has the form

GO TO (lab_1, ..., lab_n), x

(3) For some labels lab_1, lab_2, and lab_3, and for some expression exp, st is an arithmetic IF statement of the form:

IF (exp) lab_1, lab_2, lab_3

and x is used as a subscript in exp.

(4) For some symbolic name subr and expressions a_1, ..., a_n, st has the form

CALL subr(a_1, ..., a_n)

and x is used as a subscript in some a_i.

(5) For some statement st_2 and expression exp, st is a logical IF statement of the form

IF (exp) st_2

and either x is used as a subscript in exp or x is used on the second level in st_2.

Definition. label function. f is a label function for seq with respect to s if and only if for some integer n, seq is a sequence of n statements with respect to s, f is a one-to-one function, each member of the domain of f is a label, each member of the range of f is an integer greater than 0 and less than or equal to n, and for each label x in the domain of f, the (f x)th member of seq is an executable statement with respect to s.

Note. A label function is the formal device by which we associate statement labels with some of the executable statements in a program. For example, imagine that we have in mind a FORTRAN function or subroutine (i.e., a "procedure subprogram"). Suppose that the 10th, 20th, and 30th statements of the program are the only statements labeled and that the labels are 1000, 2000, and 3000 respectively. Then this FORTRAN subprogram is conveniently characterized by three mathematical objects: the syntactic environment s with respect to which the expressions and statements in the program are formed, the sequence of statements seq comprising the program, and the label function that maps 1000 to 10, 2000 to 20, and 3000 to 30 and is undefined elsewhere. We will use such triples to characterize the subprograms in our subset.

Definition. subprogram. A triple <s seq labs> is a subprogram if and only if each of the following statements is true:

(1) s is a syntactic environment.

(2) seq is a sequence of statements with respect to s.

(3) labs is a label function for seq with respect to s.

(4) The first statement of seq is a SUBROUTINE or FUNCTION statement and no other statement of seq is a SUBROUTINE or FUNCTION statement.

(5) For all ap, a, i, and d, if ap is a member of the array patterns of s, a is the name of ap, i is 1, 2, or 3, the ith member of the dimension list of ap is d, and d is a symbolic name, then both a and d are arguments of the first statement of seq.

(6) If the first statement of seq is a SUBROUTINE statement, then for each pattern p in the array, variable, label variable, statement function, or function patterns of s, the name of p is declared in exactly one type statement of seq and it is declared there to have the type of p.

(7) If the first statement of seq is a FUNCTION statement, then for each pattern p in the array, label variable, statement function, or function patterns of s, or in the variable patterns of s except for the pattern whose name is the name of the first statement of seq, the name of p is declared in exactly one type statement of seq, it is declared there to have the type of p, and the name of the first statement of seq is not declared in any type statement of seq.

(8) The name of each member of the variable patterns of s is used as a variable (other than as an actual) in some statement of seq other than a type or FUNCTION statement.

(9) The name of each member of the variable patterns of s that is an argument of the first statement of seq is used as a variable (other than as an actual in a function reference) in some statement of seq other than a type, CALL, FUNCTION, or SUBROUTINE statement of seq.

(10) The name of each intrinsic member of the function patterns of s occurs in some nontype statement of seq.

(11) If f is the name of a member of the function patterns of s and f is not the name of an intrinsic function pattern, then f is declared to be EXTERNAL in exactly one statement of seq.

(12) Each member of the block names of s is declared as a COMMON block in exactly one statement of seq.

(13) No symbolic name is declared to be in a COMMON block in two COMMON statements of seq.

(14) No argument of the first statement of seq is declared to be in a COMMON block by a COMMON statement of seq.

(15) If the first statement of seq is a FUNCTION statement, then the name of that statement is not declared to be in a COMMON block by a COMMON statement of seq.

(16) The name of each member of the statement function patterns of s is defined in exactly one statement function statement of seq.

(17) The name of each member of the array patterns of s is declared in exactly one DIMENSION statement of seq.

(18) Every label used in a GO TO assignment, unconditional GO TO, computed GO TO, assigned GO TO, arithmetic IF, or logical IF statement of seq is a member of the domain of labs.

(19) The name of each member of the subroutine patterns of s is called in at least one CALL statement of seq.

(20) The name of each member of the label variable patterns of s is used as a label variable in at least one statement of seq.

(21) At least one member of seq is a RETURN statement.

(22) The subsequence of seq obtained by deleting the first and last statements of seq satisfies the following conditions: (a) the type statements precede the other statements, (b) the COMMON, EXTERNAL, and DIMENSION statements precede the statement function and executable statements, and (c) the statement function statements precede the executable statements.

(23) For each stmt and fn, if stmt is a statement function statement of seq with respect to s, the body of the statement has a subexpression that is a function reference to fn with respect to s, and fn is the name of a statement function pattern of s, then the statement function statement of seq that defines fn precedes stmt in seq.

(24) The next to last statement of seq is an unconditional GO TO, computed GO TO, assigned GO TO, arithmetic IF, RETURN, or STOP statement.

(25) The last statement of seq is an END statement and no other statement of seq is an END statement.

If <s seq labs> is a subprogram, then the name of <s seq labs> is the name of the first statement of seq and the arguments of <s seq labs> are the arguments of the first statement of seq.

Definition. subroutine subprogram. <s seq labs> is a subroutine subprogram if and only if <s seq labs> is a subprogram and the first member of seq is a SUBROUTINE statement.

Definition. function subprogram. <s seq labs> is a function subprogram if and only if <s seq labs> is a subprogram and the first member of seq is a FUNCTION statement.

Note. We now define "superficial context." Intuitively, a superficial context is a sequence of subprograms such that each of the functions and subroutines used by any of them is defined earlier in the sequence, and such that certain "interprogram" relationships exist between the subprograms.

Definition. superficial context. c is a superficial context if and only if c is a (possibly empty) sequence of subprograms $\langle\langle s_1\ seq_1\ labs_1\rangle \ldots \langle s_n\ seq_n\ labs_n\rangle\rangle$ and each of the following statements is true:

(1) For each s_i and for each <sub v_1 ... v_n> in the subroutine patterns of s_i, there exists an integer j greater than 0 and less than i such that the first statement of seq_j is a SUBROUTINE statement, sub is the name of the first statement of seq_j, the first statement of seq_j has n arguments, and for each integer k greater than 0 but less than or equal to n, the kth argument of the first statement of seq_j is the name of v_k, and v_k is a member of either the array or variable patterns of s_j.

(2) For each s_i and for each <f t v_1 ... v_n> in the function patterns of s_i, either the pattern is an intrinsic function pattern or there exists an integer j greater than 0 and less than i such the first statement of seq_j is a FUNCTION statement, f is the name of the statement, t is the type of the statement, the statement has n arguments, and for all k greater than 0 but less than or equal to n, the kth argument of the first statement of seq_j is the name of v_k, and v_k is a member of either the array or variable patterns of s_j.

(3) If sub_i and sub_j are distinct members of c, then the name of sub_i is not the name of sub_j or the name of any intrinsic function pattern.

(4) If <s seq labs> is a member of c, then for no member sub_j of c is the name of sub_j a member of the block names of s.

(5) If <s_i seq_i $labs_i$> and <s_j seq_j $labs_j$> are distinct members of c, and if n is a member of the block names of s_i and s_j, then the COMMON statement of seq_i that declares n to be a COMMON block is identical to the COMMON statement of seq_j that declares n to be a COMMON block. Furthermore, if a is a symbolic name that is declared in seq_i to be in the COMMON block n, then there exists a pattern whose name is a that is either both a member of the variable patterns of s_i and the variable patterns of s_j or is both a member of the array patterns of s_i and the array patterns of s_j.

(6) If <s_i seq_i $labs_i$> and <s_j seq_j $labs_j$> are members of c and a is a symbolic name that is declared to be in a COMMON block n by a member of seq_i and is declared to be in a COMMON block m by a member of seq_j, then n is m. Furthermore, a is not the name of any member of c nor a member of the block names of any member of c.

Note. In Section V we comment upon some of the interprogram relationships we impose in addition to those of FORTRAN.

Note. In order to specify subprograms it is necessary to be able to refer unambiguously to variables and arrays -- even variables and arrays not declared in the subprogram itself. For example, suppose subroutine MULT declares A as a "local" array (i.e., A is declared in MULT but not in a COMMON). Suppose that MULT CALLs the subroutine TEST, which declares A to be in COMMON block BLK and modifies that "global" A. A reference to A in a FORTRAN statement in MULT is understood to be a reference to the "local" A. It is not possible for a FORTRAN statement in MULT to refer to the "global" A of TEST. However, since the value of the "global" A may affect the computation of TEST and thus the computation of MULT, it may be necessary in specifying MULT to refer to the "global" A. To permit clear talk about such matters we now define the "local" and "global" names of a subprogram and introduce the notion of the "long" and "short" names of variables and arrays. An assertion in MULT can refer to the value of the A in COMMON block BLK by using the name BLK-A, which is the "long" name of the "global" A.

Definition. local names. n is a local name of a subprogram <s seq labs> if and only if n is the name of a member of the array, variable, or label variable patterns of s, and n is not declared by any COMMON statement of seq to be in any COMMON block.

Definition. COMMON names. n is a COMMON name of a subprogram <s seq labs> if and only if for some b and some v, b is a member of the block names of s, v is declared by some COMMON statement of seq to be in the COMMON block b, and n has the form

b-v

Definition. short name. n is the short name of m if and only if n is a symbolic name and either (a) m is a symbolic name and n is m or (b) for some symbolic name b, m has the form b-n.

Definition. long name. n is the long name of m with respect to sub if and only if sub is a subprogram and either (a) n is a local name of sub and n is m or (b) n is a COMMON name of sub and m is the short name of n.

Example. Suppose we have in mind a subprogram sub in which ARRAY is declared as an array but is not in COMMON, and SIZE is declared as a variable in the COMMON block named BLK. Then ARRAY is a local name of sub. BLK-SIZE is a COMMON name of sub. The short name of ARRAY is ARRAY. The short name of BLK-SIZE is SIZE. The long name of ARRAY with respect to sub is ARRAY. The long name of SIZE with respect to sub is BLK-SIZE.

Note. We now define the "global names" of a subprogram and the "global sort" of a long name. Intuitively, the global names of a subprogram are the global variables and arrays of the subprogram and all the subprograms it CALLs. Of course, the unambiguous long names of the variables are used.

Definition. global names. If c is a superficial context and sub is a member of c, then the global names of sub with respect to c are the COMMON names of sub together with the global names of each sub_j of c such that the name of sub_j is the name of a nonintrinsic function or subroutine pattern of sub.

Definition. global sort. t is the global sort of n in sub with respect to c if and only if c is a superficial context, sub is a member of c, and one of the following is true:

(1) n is a local name of sub and t is the sort of the array, variable, or label variable pattern in sub with name n.

(2) n is a global name of sub with respect to c, n has the form b-v for some b and v, and t is the sort of the array or variable pattern with name v in any member of c in which v is declared to be in block b.

Note. We next define the notion that a variable or an array is "possibly smashed" by a subprogram. The intuitive idea is that v is possibly smashed in a subprogram if execution of the subprogram appears sometimes to alter the value of v either by assigning to it or by CALLing a subroutine that possibly smashes it. The notion of "possibly smashed" is used in a variety of places in this document. For example, it is involved in the question of whether a subroutine CALL violates the FORTRAN prohibition against aliasing. The notion is also used to extend

the output assertion of a subprogram by the implicit assertion that variables not possibly smashed are unmodified by CALLs of the subprogram.

There is one subtle aspect to the definition of "possibly smashed." Although we check that a variable might be smashed by the execution of a subroutine subprogram, we do not check that it is possibly smashed by the execution of a function subprogram. The reason is that when we define a "syntactically correct context" we will require that function subprograms not possibly smash any variables except locals that are not arguments (i.e., functions have no side effects). Thus, provided one is dealing with a "syntactically correct context," our definition of "possibly smashed" indeed guarantees that no variable of the calling program is modified by the execution of a function subprogram.

Definition. possibly smashed. v is possibly smashed by $\langle s_i\ seq_i\ labs_i \rangle$ in c through st if and only if (a) c is a superficial context, (b) $\langle s_i\ seq_i\ labs_i \rangle$ is a member of c, (c) st is a statement of seq_i, and (d) for some u, u is the short name of v, v is a global name or a local name of $\langle s_i\ seq_i\ labs_i \rangle$, and one of the following statements is true:

(1) v is a local or COMMON name of $\langle s_i\ seq_i\ labs_i \rangle$, and for some expressions x_1, x_2, x_3, and e with respect to s_i, st has of one of the forms

    u = e

    $u(x_1)$ = e

    $u(x_1, x_2)$ = e

    $u(x_1, x_2, x_3)$ = e

(2) v is a local name of $\langle s_i\ seq_i\ labs_i \rangle$, and for some k, st has the form

    ASSIGN k TO u

(3) For some symbolic name sub, and expressions e_1, ..., and e_n with respect to s_i, st has the form

    CALL sub

or

CALL sub(e_1, ..., e_n)

and for the member <s_j seq_j $labs_j$> of c with the name sub, either (a) v is a global name of <s_j seq_j $labs_j$> and v is possibly smashed by <s_j seq_j $labs_j$> through some member of seq_j or (b) v is a local or COMMON name of <s_i seq_i lab_i>, and for some i greater than 0 and less than n+1, e_i is a variable reference or array reference to u, and the ith argument of <s_j seq_j $labs_j$> is possibly smashed by <s_j seq_j $labs_j$> in c through some member of seq_j.

(4) For some statement st_2 and expression exp, st is a logical IF statement of the form

IF (exp) st_2

and v is possibly smashed by <s_i seq_i $labs_i$> in c through st_2.

[If we say "v is possibly smashed by sub in c" (omitting "through st"), we mean "for some st, v is possibly smashed by sub in c through st."]

Note. We complete the syntactic characterization of our FORTRAN subset by defining a "syntactically correct context" to be a sequence of subprograms that, in addition to being a superficial context, contains no function subprogram that causes side effects and no CALL statement that violates certain aliasing restrictions. In Section V we comment upon these restrictions and their relationship to FORTRAN.

Definition. syntactically correct context. c is a syntactically correct context if and only if c is a superficial context and each of the following statements is true:

(1) For each function subprogram sub of c, no global name of sub nor argument of sub is possibly smashed by sub in c.

(2) In each subroutine subprogram <s seq labs> of c, no argument of <s seq labs> that is a member of the dimension list of an array pattern of s is possibly smashed by <s seq labs> in c.

(3) For each <s seq labs> and sub_j in c, for each sub, for each sequence of expressions <e_1 ... e_n> with respect to s, and for each statement in seq of the form

CALL sub(e_1, ..., e_n)

if sub is the name of sub_j, then for each integer i greater than 0 and less than or equal to n, if the ith argument of sub_j is possibly smashed by sub_j in c, then for some symbolic name v each of the following is true:

(a) e_i is either a variable reference to v or an array reference to v,

(b) the long name of v with respect to <s seq labs> is not a global name of sub_j, and

(c) for each integer j greater than 0, not equal to i, and less than or equal to n, e_j is not a variable, array, or array element reference to v.

(4) For each <s seq labs> and sub_j in c, for each m, sub, and sequence of expressions <e_1 ... e_n> with respect to s, and for each statement in seq of the form

CALL sub(e_1, ..., e_n)

if sub is the name of sub_j and m is a COMMON name of <s seq labs> that is possibly smashed by sub_j in c, then for k greater than 0 and less or equal to n, e_k is not a variable, array, or array element reference to the short name of m.

(5) For each <s seq labs> in c, if x is a COMMON or local name of <s seq labs> and the short name of x is used on the second level in some statement of seq, then x is not possibly smashed by <s seq labs> in c through any CALL statement of <s seq labs>.

VII FLOW GRAPHS

In this section we describe the flow of control through a FORTRAN subprogram.

Informally, the flow is specified by an "ordered, directed graph" whose nodes are associated with the statements of the subprogram. A proof of correctness involves the consideration of all "paths" through the graph. To aid the consideration of the possibly infinite number paths created by loops, the exploration process utilizes a "cover" of the graph, which is a subset of the nodes sufficient to "cut" every loop. Given a graph and a cover for it, it is then possible to identify a finite number of paths through the graph, called the "Floyd paths," such that the correctness of these paths implies the correctness of all paths through the graph.* We make these graph theoretic terms precise before discussing the graphs for subprograms.

Definition. ordered, directed graph. <n e> is an ordered directed graph if and only if n is a set and e is a function whose range is a subset of n and whose domain is a subset of the Cartesian product of n with the positive integers. Each member <<x i> z> of e is an edge of the graph, x is the head of the edge, z is the tail of the edge, and <<x i> z> is the ith edge leading from x.

Definition. path. If g is an ordered, directed graph, then p is a path in g if and only if p is a (possibly empty and possibly infinite) sequence $\langle e_1\ e_2\ \ldots\rangle$ of edges of g and for each member e_{i+1} of p, if i>0 then the tail of e_i is the head of e_{i+1}.

Definition. cover. If g is an ordered, directed graph, then c is a cover for g if and only if c is a subset of the nodes of g and for each infinite path p in g the tail of some element of p is a member of c.

* Readers interested in a more tutorial sketch of the application of the Floyd method should see [5].

Definition. Floyd path. p is a Floyd path of g for c if and only if g is an ordered, directed graph, c is a cover for g, p is a nonempty, finite path in g, the head of the first and the tail of the last members of p are members of c, and for each integer i greater than 1 and less than or equal to the length of p, the head of ith member of p is not a member of c.

Definition. flow graph. The flow graph of a subprogram <s seq labs> is the pair <n e> such that each of the following is true:

(1) n is the set of all integers j such that the absolute value of j is less than or equal to the length of seq and either j is positive and the jth member of seq is an executable statement or j is negative and the -jth member of seq is a logical IF statement.

(2) e is the set of all <<i j> k> such that for some st

(a) i and k are members of n, and

(b) either i is positive and st is the ith statement of seq or i is negative and the -ith statement of seq contains st, and (in either case) one of the following statements is true:

(i) st is an assignment, GO TO assignment, CALL, CONTINUE, or PAUSE statement, j is 1, and k is |i|+1;

(ii) for some label m, st has the form

GO TO m

j is 1, and k is (labs m);

(iii) for some labels m_1, ..., and m_n, st has the form

GO TO i, (m_1, ..., m_n)

or

GO TO (m_1, ..., m_n), i

j is greater than 0 but less than or equal to n, and k is (labs m_j);

(iv) for some labels m_1, m_2, and m_3, and for some expression e, st has the form

IF (e) m_1, m_2, m_3

and j is 1, 2, or 3, and k is, respectively, (labs m_1), (labs m_2), or (labs m_3); or

(v) for some expression e and some statement st', st has the form

IF (e) st'

j is 1 or 2, and k is, respectively, -i or i+1.

Definition. statement of. If <s seq labs> is a subprogram and n is a member of the nodes of the flow graph of <s seq labs>, then the statement of n is the nth member of seq if n is positive and the statement contained in the -nth member of seq if n is negative.

Note. A node n of the flow graph g of a subprogram is the head of some edge of g if and only if the statement of n is neither a RETURN or a STOP statement.

Definition. reachable. m is reachable from n in g if and only if g is an ordered, directed graph, m and n are nodes of g, and there exists a finite path p of g such that the head of the first edge of p is n and the tail of the last edge of p is m.

Note. To enforce the ANSI 66 requirement that every executable statement in a subprogram "can" sometimes be executed, we will require that every node in the flow graph of the subprogram be reachable from the node corresponding to the first executable statement.

VIII TERMS

Before we discuss the specification of subprograms we define the logical theory in which we are operating. For example, there is an intuitive feeling that for every FORTRAN expression (e.g., (X+A(I))) there must be a term in the logic that in some sense denotes the value of that expression. The verification condition generator must construct for each FORTRAN expression the corresponding term of the logic. Therefore, at the very least we must define the well-formed terms of the logic and specify the correspondence between FORTRAN expressions and terms.

Notation. expressions versus terms. Some logicians use the words "expression" and "term" interchangeably. In this document we use the word "expression" exclusively to refer to FORTRAN expressions with respect to a given syntactic environment, as previously defined. We use the word "term" exclusively to refer to well-formed terms in the mathematical logic in which we operate.

Note. We first specify the "basic FORTRAN theory," which is a logic produced by extending the logic described in [5] and [6]. The syntax of the logic is the prefix syntax used by Church's lambda-calculus. For example, we write (MOD X Y) to denote the term that others might denote by MOD(X,Y), mod(X,Y), or X mod Y.

In addition to a version of the propositional calculus with function symbols and equality, the logic provides a principle under which new types of inductively constructed objects may be introduced, a principle under which new recursive functions may be defined, and a principle of induction.

The logic does not provide the usual notion of predicates or relations. Without loss of generality we use functions exclusively; a

property is expressed by writing down a term that is either equal to the object (FALSE) or not equal to the object (FALSE) according to whether the property fails to hold or holds. (The constant (TRUE) is a commonly used object not equal to (FALSE).) For example, the function ZLESSP, of two arguments, returns (TRUE) or (FALSE) according to whether its first argument is less than its second.

The logic does not provide explicit quantification. Variables in formulas are implicitly universally quantified. To express certain forms of quantification it is necessary to define new recursive functions. For example, to assert that some members of a finite sequence have a given property one may introduce the recursive function that maps over the sequence and checks for the property.

The basic FORTRAN theory contains some of the function symbols used in the transcription of FORTRAN expressions into mathematical terms. For example, it contains the function ZPLUS that returns the sum of its two integer arguments, and the previously mentioned ZLESSP.

Definition. basic FORTRAN theory. The basic FORTRAN theory is the logical theory obtained by extending the theory defined in [5] and [6] as specified in Appendix A.

Note. We now make some definitions that make it easier to refer to certain function symbols of the basic FORTRAN theory.

Definition. FORTRAN recognizers. The FORTRAN recognizer for a type INTEGER, REAL, DOUBLE, COMPLEX, or LOGICAL is (respectively) the function symbol ZNUMBERP, RNUMBERP, DNUMBERP, CNUMBERP, or LOGICALP.

Definition. function symbol for t and op. For the combinations of type t and operator symbol op given below, the function symbol for t and op is defined by the following table:

function symbol for t and op

op\t	INTEGER	REAL	DOUBLE	COMPLEX
+	ZPLUS	RPLUS	DPLUS	CPLUS
-	ZDIFFERENCE	RDIFFERENCE	DDIFFERENCE	CDIFFERENCE
*	ZTIMES	RTIMES	DTIMES	CTIMES
/	ZQUOTIENT	RQUOTIENT	DQUOTIENT	CQUOTIENT
.LT.	ZLESSP	RLESSP	DLESSP	
.LE.	ZLESSEQP	RLESSEQP	DLESSEQP	
.EQ.	ZEQP	REQP	DEQP	CEQP
.NE.	ZNEQP	RNEQP	DNEQP	CNEQP
.GE.	ZGREATEREQP	RGREATEREQP	DGREATEREQP	
.GT.	ZGREATERP	RGREATERP	DGREATERP	

Definition. exponentiation function symbol. The exponentiation function symbol for certain combinations of types t_1 and t_2 is defined by the following table:

t_1	t_2	exponentiation function symbol
INTEGER	INTEGER	ZEXPTZ
REAL	INTEGER	REXPTZ
DOUBLE	INTEGER	DEXPTZ
COMPLEX	INTEGER	CEXPTZ
REAL	REAL	REXPTR
REAL	DOUBLE	REXPTD
DOUBLE	REAL	DEXPTR
DOUBLE	DOUBLE	DEXPTD

Note. The next five definitions make it easier to describe certain terms in our logic.

Definition. list term and LENGTH. We define the list term for a sequence of terms $\langle s_1 \ldots s_n \rangle$ to be the term (LIST $s_1 \ldots s_n$). If t is the list term for a sequence of length n, then the LENGTH of t is n.

Definition. conjunction. The conjunction of the terms s_1, s_2, ..., s_n is the term (AND s_1 s_2 ... s_n).

Definition. disjunction. The disjunction of the terms s_1, s_2, ..., s_n is the term (OR s_1 s_2 ... s_n).

Definition. implication. If u and v are formulas, the implication from u to v is the formula (IMPLIES u v).

Definition. lexicographic comparison. If t_1 and t_2 are terms then the lexicographic comparison of t_1 with t_2 is (LEX t_1 t_2).

Note. The specification of a given FORTRAN subprogram begins with the production by the user of a "FORTRAN theory" rather than the basic FORTRAN theory. As a rule the basic FORTRAN theory is too primitive to permit the expression of the specifications of interesting programs without first being extended by the definition of new functions. For example, to specify a matrix multiplication subroutine, the user might first extend the basic FORTRAN theory by the definition of the usual mathematical operations and relations on vectors and matrices.

Definition. FORTRAN theory. A FORTRAN theory is an extension of the basic FORTRAN theory. The extension may add no axiom (e.g., definition) in which the function symbol START is used.

Note. The user supplied FORTRAN theory must be extended by the addition of several new function symbols before we arrive at the theory in which the verification conditions will be proved. For example, for each function subprogram referenced in the subprogram being verified, we will add an undefined function symbol whose value is as specified by the (previously accepted) input/output assertions for the function subprogram. These function symbols are used in the terms representing the values of FORTRAN expressions. To ensure that this extension makes sense (e.g., does not attempt to "redefine" an existing function) we make the next definition.

Definition. appropriate. T is an appropriate theory for c if and only if each of the following statements is true:

(1) T is a FORTRAN theory.

(2) c is a syntactically correct context.

(3) NEXT and BEGIN are not function symbols of T.

(4) For all <s seq labs> in c, if v is the long name of a member of the variable, label variable, or array patterns of s, then v is not the name of a function symbol of T, nor is v NEXT or BEGIN.

(5) If v is the name of a function subprogram of c, then v is not the name of a function symbol of T, nor is v NEXT or BEGIN.

Note. For reasons indicated previously, we require that no function subprogram of c have as its name the name of a function symbol of T. But certain specific function symbols are required to be in T (e.g., EQUAL, ZPLUS, ZEXPTZ) and these symbols are acceptable FORTRAN names. Thus, this restriction technically prevents the use of our system to verify FORTRAN programs involving functions with those "built-in" names because no "appropriate theory" exists. Such problems are easily avoided by systematic renaming of the mathematical functions used to denote the values of FORTRAN entities. However, to avoid making this document even more obscure, we here agree to live with the naming limitations imposed above.

Given a theory T appropriate for a syntactically correct context c, we now describe three incremental extensions of T, the last of which is the theory in which one must prove the verification conditions for a subprogram of c. The three extensions are obtained by adding certain new function symbols, including one for each global name and argument, local name, and nonintrinsic function subprogram used in the subprogram being verified. By producing the final theory incrementally and giving names to the intermediate versions we make it possible to say "x is a term" of one of these intermediate theories, which is a convenient way of saying that x contains no function symbols except those in the specified theory.

Definition. primary verification extension. T_2 is the primary verification extension of T_1 for c and sub_j if and only if T_1 is an appropriate theory for c, sub_j is a member of c, and T_2 is the extension of T_1 that results from adding as a function symbol of one argument each of the global names of sub_j and each of the arguments of sub_j.

Definition. secondary verification extension. T_2 is the secondary verification extension of T_1 for c and sub_j if and only if T_1 is an appropriate theory for c, sub_j is a member of c, and T_2 is the extension

of the primary verification extension of T_1 that results from adding as a function symbol of one argument each of the local names of sub_j that is not an argument of sub_j.

Definition. tertiary verification extension. T_2 is the tertiary verification extension of T_1 for c and sub_j if and only if T_1 is an appropriate theory for c, sub_j is a member of c, and T_2 is the extension of the secondary verification extension of T_1 that results from both of the following:

(1) Adding NEXT and BEGIN as function symbols of 1 and 0 arguments respectively.

(2) Adding as a function symbol the name of each function pattern of the syntactic environment of sub_j that is not an intrinsic function pattern and by providing each such function symbol f with the number of arguments that is the sum of (a) the number of arguments of the function pattern with name f and (b) the number of global names of the member of c whose name is f.

Note. If ARRAY is a local name of a subprogram, our convention is to denote the value of ARRAY at a given point in the execution of the subprogram by a term of the form (ARRAY state), where state is a constant term that may be thought to denote (in a completely arbitrary way) the state of the processor. The only use of states is to permit us to use terms such as (ARRAY state) to refer to the various values taken on by program variables during execution. We formalize the notion of "state term" later. To refer to the current value of ARRAY in an invariant, the user writes (ARRAY STATE). It is understood that the current state term will be substituted for STATE when the invariant is encountered by the verification condition generator. It is important that STATE not occur arbitrarily within an invariant since that would permit the invariant to exploit the structure of what amounts only to a naming convention. The next definition provides us with a succinct way of saying that some particular set of terms occur only as arguments to those functions denoting the values of program variables.

Definition. incarcerates. p incarcerates v with respect to c and sub_j if and only if c is a syntactically correct context, sub_j is member

of c, v is a set, and for some theory T appropriate for c, p can be obtained from some term p′ of T that does not have any member of v as a subterm by simultaneously replacing variables with terms of the form (f v′) where f is a global or local name of sub_j and v′ is a member of v.

Example. Suppose T is a theory appropriate for c and c contains some subprogram sub_j and CNT is a local name of sub_j. Then the term

```
(ZEQP (CNT NEWSTATE)
      (ZPLUS 1 (CNT STATE)))
```

incarcerates ⟨NEWSTATE, STATE⟩ with respect to c and sub_j, because the term can be obtained by instantiating

```
(ZEQP U
      (ZPLUS 1 V))
```

by replacing U with (CNT NEWSTATE) and V with (CNT STATE). Since the following term cannot be obtained by such an instantiation, it does not incarcerate ⟨NEWSTATE STATE⟩:

```
(ZEQP (CNT NEWSTATE)
      (ZPLUS STATE (CNT STATE)))
```

Note. We now formally define the mapping from FORTRAN expressions to terms in a FORTRAN theory.

Definition. statification. If c is a syntactically correct context, <s seq labs> is a member of c, and e is an expression with respect to s, then the statification of e (denoted [e] when c and <s seq labs> are obvious from context) is defined, inductively, as follows.

(1) If e is a constant then
    if e is .TRUE., then [e] is (TRUE),
    if e is .FALSE., then [e] is (FALSE),
    and otherwise [e] is e.

(2) If e is a token, then [e] is (e).

(3) If e is a variable or array reference, then [e] is the term (e′ STATE), where e′ is the long name of e with respect to <s seq labs>.

(4) If e is an array element reference, then
for each v and expression x_1, if e has the form $v(x_1)$,
then [e] is (ELT1 [v] [x_1]),
for each v and for all expressions x_1 and x_2,
if e has the form $v(x_1, x_2)$,
then [e] is (ELT2 [v] [x_1] [x_2]), and
for each v and for all expressions x_1, x_2, and x_3,
if e has the form $v(x_1, x_2, x_3)$,
then [e] is (ELT3 [v] [x_1] [x_2] [x_3]).

(5) If e is an arithmetic or relational expression with argument sequence <x_1 x_2> and operation symbol op, op is not **, the sort of x_1 and x_2 is <t>, and f is the function symbol for t and op, then [e] is (f [x_1] [x_2]).

(6) If e is an arithmetic expression with operation symbol ** and argument sequence <x_1 x_2>, the sort of x_1 is <t_1>, the sort of x_2 is <t_2>, and f is the exponentiation function symbol for t_1 and t_2, then [e] is (f [x_1] [x_2]).

(7) For all expressions x_1 and x_2, if e has one of the forms

(x_1 .AND. x_2)
(x_1 .OR. x_2)
(.NOT. x_1)

then [e] is

(AND [x_1] [x_2])
(OR [x_1] [x_2])
(NOT [x_1])

respectively,

(8) For each f that is the name of a statement function pattern of s and for all expressions x_1, ..., and x_n, if e has the form

$f(x_1, \ldots, x_n)$

then [e] is the statification of the result of simultaneously replacing each of the arguments of the definition of f in seq with the corresponding x_i in the body of the definition of f in seq.

(9) For each f that is the name of an intrinsic function pattern and for all expressions x_1 ..., and x_n, if e has the form

$f(x_1, \ldots, x_n)$

then [e] is (f [x_1] ... [x_n])

(10) Finally, for each f that is the name of some nonintrinsic function pattern of s and for all expressions x_1, ..., and x_n, if e has the form

$$f(x_1, \ldots, x_n),$$

then [e] is

$$(f\ [x_1] \ldots [x_n]\ (k_1\ \text{STATE}) \ldots (k_m\ \text{STATE}))$$

where the k_i are the global names, in alphabetical order, of the member of c whose name is f.

Example. Suppose PROD is the name of a REAL valued function subprogram, subfn, of c. Suppose that subfn takes one argument and declares A1 and A2 to be in COMMON block BLK. Suppose further that BLK-A1 and BLK-A2 are the only global names of subfn. Finally, suppose that VECT is a one-dimensional array declared as a local name in some other subprogram, sub, of c, that I is declared in sub to be an INTEGER variable in COMMON block TEMP, that MAX is a local name of sub, and that

```
(PROD(MAX)+VECT(I))
```

is an expression with respect to the syntactic environment of sub. Then the statification of the above expression is:

```
(RPLUS (PROD (MAX STATE)
             (BLK-A1 STATE)
             (BLK-A2 STATE))
       (ELT1 (VECT STATE) (TEMP-I STATE))).
```

Notation. [x, y] is the result of replacing the variable STATE with y in the statification of x.

Definition. term substitution. A finite set s of ordered pairs is said to be a term substitution provided that for each ordered pair <u v> in s, u and v are terms and no other member of s has u as its first component. The result of applying a term substitution s to a term t is recursively defined as follows:

For each v, if <t v> is a member of s,
   the result is v;
else if t is a variable, the result is t;
else t is of the form $(f\ t_1 \ldots t_n)$
   and the result is $(f\ t_1' \ldots t_n')$,

where t_i' is the result of applying the term substitution s to t_i.

Example. The result of applying the term substitution <<X (G)> <(F X) (H X)>> to the term (PLUS X (F X)) is the term (PLUS (G) (H X)). In particular, the result is not (PLUS (G) (F (G))).

IX SPECIFIED CONTEXTS

We now begin discussing the specification of the subprograms of a syntactically correct context. We first formalize the notion of specifying the input and output assertions of the subprograms. Intuitively, if a subprogram has been proved correct, then whenever it is legally invoked (e.g., by a CALL statement we consider syntactically and semantically correct) and the input assertion is true just before the execution of the first statement (that is, after the association of the actuals with the formals), then in the execution of the subprogram a RETURN will eventually be executed, and at the inception of the RETURN statement the output assertion will be true.

Definition. specification for a context. A pair of functions <inpt outpt> is a specification for c and T if and only if each of the following statements is true:

(1) T is an appropriate theory for c.

(2) The domain of inpt is the set of members of c and for each member sub of c, the value, v, of inpt on sub is a term in the primary extension of T for c and sub, and v incarcerates {STATE} with respect to c and sub.

(3) The domain of outpt is the set of members of c, and for each member sub of c, the value, v, of outpt on sub is a term in the primary extension of T for c and sub, and

   (a) if sub is a subroutine subprogram, then v incarcerates {STATE, NEWSTATE} with respect to c and sub, and for every subterm of v of the form (f NEWSTATE), f is possibly smashed by sub in c, and

   (b) if sub is a function subprogram, then v contains no variables except STATE and ANS and v incarcerates {STATE} with respect to c and sub.

Definition. input assertion and output assertion. Suppose we have in mind a given inpt, outpt, c and T, such that <inpt outpt> is a

specification for c and T. If sub is a member of c, we define the input assertion of sub to be the formula that is the value of inpt on sub, and we define the output assertion of sub to be the formula that is the value of outpt on sub.

Note. We now formalize the notion of attaching to certain statements in a subprogram the "Floyd invariants." Intuitively, these invariants are assertions about the values of program variables and are supposed to be true every time the processor encounters the statement.

In addition to proving that each subprogram meets its input/output specification, provided it terminates, we desire to prove that each subprogram terminates when called in an environment satisfying its input assertion. To do this we require the user to specify the amount of "time" the program will run by attaching "clocks" to various statements. Intuitively, the "input clock" says how long the program will run (as a function of the initial environment) and the interior clocks say how much time remains (as a function of the variables in the current state). The verification conditions force us to prove that each time a clock is encountered, strictly less "time" remains on it than on the previously encountered clock. Provided the "less than" relation used is well founded, proof of the clock verification conditions is sufficient to imply termination of the program.

It is often convenient for clocks to be functions into the natural numbers and to be compared with the well-founded Peano "less than" relation. However, such a scheme makes it difficult to prove termination for programs involving nested loops because the necessary clocks often involve multiplication, exponentiation, and so on. To mitigate this problem somewhat we make the convention that each clock be an n-tuple of natural numbers (for some value of n fixed by the user for a given subprogram), and we compare these n-tuples with the function LEX, defined in the basic FORTRAN theory.

For example, to prove the termination of a nested loop, the outer of which counts I down somehow while changing J arbitrarily, and the inner of which counts J down while holding I fixed, the clock (LIST (I STATE) (J STATE)) suffices.

Definition. annotation. A sequence of three elements <inpclk lpinv lpclk> is called an annotation with respect to T, c, and sub if and only if each of the following statements is true:

(1) T is an appropriate theory for c.

(2) sub is a member of c.

(3) inpclk is the list term for some sequence of terms in the primary extension of T for c and sub, inpclk has no variable subterm, and inpclk incarcerates {(START)} with respect to c and sub.

(4) lpinv is a function and for some w, w is the domain of lpinv, w is a subset of the nodes of the flow graph of sub, the statement of each member of w is a CONTINUE statement, w covers the flow graph of sub, and for each member r of the range of lpinv, r is a term of the secondary extension of T for c and sub and r incarcerates {STATE, (START)} with respect to c and sub.

(5) lpclk is a function whose domain is the domain of lpinv and for each member r of the range of lpclk, r is the list term of a sequence of terms in the secondary extension of T for c and sub, r and inpclk have the same LENGTH, r incarcerates {STATE, (START)} with respect to c and sub, and has no variable subterm except STATE.

Convention. For the remainder of this section, let us fix upon a T, c, sub, s, seq, labs, inpt, outpt, inpclk, lpinv, and lpclk such that each of the following statements is true:

(1) T is an appropriate theory for c.

(2) <s seq labs> is a member of c and sub is <s seq labs>.

(3) <inpt outpt> is a specification of c.

(4) <inpclk lpinv lpclk> is an annotation for sub.

Definition. input clock, loop invariant, and loop clock. The input clock is inpclk. If node is a member of the domain of lpinv, then the loop invariant for node is the value of the function lpinv applied to node, and the loop clock for node is the value of the function lpclk applied to node.

Note. We now introduce the ideas of the "partially instantiated" input and output assertions. These assertions are attached to the entrances and exits of the subprogram.

Definition. partially instantiated input assertion. If sub is a function subprogram, then the partially instantiated input assertion is the conjunction of (a) the result of substituting (START) for STATE in the input assertion of sub and (b) the term (DEFINEDP (a (START))) for each a that is an argument of sub and is not the name of an array pattern in s. If sub is a subroutine subprogram, then the partially instantiated input assertion is the conjunction of (a) the result of substituting (START) for STATE in the input assertion of sub and (b) the term (DEFINEDP (a (START))) for each a that is both an argument of sub and a member of the dimension list of some array pattern of s.

Note. (START) is the arbitrarily chosen constant denoting the state at the beginning of the execution of the subprogram being verified.

We require that all nonarray arguments to function subprograms be defined, and we automatically extend the user's input assertion to that effect. FORTRAN does not have such a requirement. We have already adopted the requirement that no function subprogram have side effects, including side effects on arguments. Thus, an undefined argument is useless: it cannot be referenced in the subprogram until it is smashed by an assignment or CALL, and it cannot be smashed. The question of the definedness of arrays never comes up. Instead, one is interested in the definedness of particular elements of arrays. Rather than automatically extend every function's input assertion with the draconian requirement that every element of every array be defined, we put no built-in requirements on arrays and thus force the user to state in his input assertions whatever hypotheses are needed about particular array elements.

Definition. partially instantiated output assertion. If sub is a function subprogram, fn is the name of the first statement of seq, and p is the output assertion of sub, then the partially instantiated output assertion is the result of substituting (START) for STATE and (fn STATE) for ANS in

```
(AND (DEFINEDP ANS)
     p)
```

If sub is a subroutine subprogram, then the partially instantiated output assertion is the result of substituting (START) for STATE and STATE for NEWSTATE in the output assertion of sub.

Note. NEWSTATE in an output assertion of a subprogram being verified is understood to refer to the state at the inception of the RETURN statement. That is, on each exit path from the subprogram the verification condition produced will conclude with the partially instantiated output assertion, further instantiated by the replacement of NEWSTATE by the state term at the end of the path.

The user writes the variable ANS in the output assertion of a function subprogram when he wishes to refer to the value delivered by the subprogram. The FORTRAN convention for defining the value of a function is to assign the desired result to the variable with the same name as the function. Thus, in the theory in use when we verify a function subprogram with name fn, fn is a function of one argument, the state of the processor, and denotes the value of a FORTRAN variable. For the verification of fn we replace ANS in the output assertion by (fn NEWSTATE).

Consider what happens when we have verified fn and are now producing the verification conditions for a subprogram that calls fn. In the theory in use when we verify a program that calls fn, fn is a function of n+m arguments (where n is the number of arguments of the FORTRAN subprogram fn and m is the number of its global names). When we use the output assertion of fn to constrain the value delivered by the expression $fn(e_1, \ldots, e_n)$, we replace ANS by $[fn(e_1, \ldots, e_n)]$ and then replace STATE by the current state term.

The next definition makes it more convenient to describe the verification condition generator. It is useful to extend the flow graph of the subprogram by imagining the insertion into the program of a CONTINUE statement immediately preceding what was previously the first

executable statement of the program. We then attach to this statement the input assertion and input clock of the subprogram.

Definition. extended flow graph. The extended flow graph of sub is the ordered, directed graph whose nodes are the nodes of the flow graph of sub together with one additional node, 0, and whose edges are the edges of the flow graph of sub together with the edge <<0 1> i>, where i is the least positive integer such that the ith member of seq is an executable statement.

Definition. statement of 0. The statement of 0 is CONTINUE.

Note. Given the extended flow graph, we can now combine the notions of the input assertion and loop assertions and input clock and loop clocks to get, simply, the "assertions" and "clocks."

Definition. assertion, clock, and decorated nodes. Let g be the the extended flow graph of sub. Below we define the decorated nodes of g, the assertion for a decorated node, and the clock for some of the decorated nodes. A node n of g is in the decorated nodes of g if and only if one of the following is true:

(1) n is 0, in which case the assertion for n is the partially instantiated input assertion and the clock for n is the input clock.

(2) n is a member of the domain of lpinv, in which case the assertion for n is the loop invariant for n and the clock for n is the loop clock for n.

(3) The statement of n is a STOP statement, in which case the assertion for n is (FALSE), and the clock for n is undefined.

(4) The statement of n is a RETURN statement, in which case the assertion for n is the partially instantiated output assertion and the clock for n is undefined.

X THE VERIFICATION CONDITIONS

We are finally ready to describe what one has to prove to verify a FORTRAN subprogram. First we formally define the arbitrary constants used to denote "states." Then we spell out the "global assumptions" that may be used in the proof of the verification conditions (e.g., the input assertion and the type of values taken on by the functions denoting variables). Next we discuss the handling of FORTRAN expressions: what may be assumed about the value of an expression (e.g., that the output assertion is true of the value returned by a function subprogram) and what must be proved about an expression (e.g., that the input assertion for a function subprogram holds). Then we turn to FORTRAN statements: what may be assumed about the change of state induced by the execution of a statement (e.g., that a CALL statement changes the state as specified by its output assertion) and what must be proved about a statement (e.g., that the input assertion for a subroutine holds). Finally, we combine all these concepts to say what must be proved about the paths through the program.

Convention. For the remainder of this section, let us fix upon the T, c, sub, s, seq, labs, inpt, outpt, inpclk, lpinv, and lpclk agreed upon in the previous section.

Definition. PATHS. Suppose g is the extended flow graph of sub. Then PATHS is the set of Floyd paths of g for the decorated nodes of g.

Definition. state term. For each path p in PATHS and for each edge e in p, we define the state term of e inductively, as follows. For the first edge of p, the state term is (START) if the head of the first edge is 0; otherwise the state term of the edge is (BEGIN). For each noninitial edge e of p, let stp be the state term of the preceding edge. The state term of e is the term (NEXT stp) if the head of e is an assignment, GO TO assignment, or CALL statement and is stp otherwise.

Note. Thus, as previously noted, (START) is the term denoting the state at the beginning of the execution of sub. (BEGIN) is the arbitrarily chosen term denoting the state at the beginning of any interior path. Every time we move past an assignment or CALL statement (the only two statement types that cause state changes) the state is "bumped" by applying the undefined function NEXT. Thus, on an interior path containing two assignments and a CALL statement, the state at the beginning is (BEGIN) and the state at the end is (NEXT (NEXT (NEXT (BEGIN)))). The only information relating stp and (NEXT stp) is that provided by the semantics of assignment and the output assertions of subroutines.

We next define the set of assumptions that may be used in the proofs of all of the verification conditions.

Definition. global assumptions. a is a member of the global assumptions if and only if one of the following is true:

(1) a is the partially instantiated input assertion.

(2) For some local or global name n of sub and for some <t d_1 ... d_k>, <t d_1 ... d_k> is the global sort of n, t′ is the FORTRAN recognizer for t, and a has one of the forms

```
(IMPLIES (DEFINEDP (n STATE))
         (t' (n STATE))),

(IMPLIES (DEFINEDP (ELT1 (n STATE) I))
         (t' (ELT1 (n STATE) I))),

(IMPLIES (DEFINEDP (ELT2 (n STATE) I J))
         (t' (ELT2 (n STATE) I J))),

(IMPLIES (DEFINEDP (ELT3 (n STATE) I J K))
         (t' (ELT3 (n STATE) I J K)))
```

according to whether <t d_1 ... d_k> is of length 1, 2, 3, or 4.

Example. If ARRAY is a local name of sub and is declared as a two-dimensional array of type INTEGER, then one of the global assumptions is that if the I,Jth element of ARRAY is DEFINEDP, then it is an integer (e.g., recognized by ZNUMBERP).

Note. Now we spell out what may be assumed about the value of a FORTRAN expression. For example, in proving that the input assertion for a subroutine CALL is satisfied, we get to assume the "output assumptions" for the arguments of the CALL.

Definition. output assumption. We now define inductively the concept of the output assumption for an expression e with respect to a state term stp. Throughout the definition, all output assumptions are understood to be with respect to stp.

(1) If e is a constant, a token, or a variable or array reference, the output assumption for e is (TRUE).

(2) If e is an array element reference, the output assumption for e is the conjunction of the output assumption for each member of the subscript sequence of e.

(3) If e is an arithmetic, relational, or logical expression, or a function reference to an intrinsic function, the output assumption for e is the conjunction of the output assumption for each member of the argument sequence of e.

(4) If e is a function reference to a statement function f with argument sequence $\langle x_1 \ldots x_n \rangle$ then the output assumption for e is the output assumption for the expression that results from substituting into the body of the definition of f the expressions $x_1, \ldots, x_n$ for the corresponding arguments of the definition of f.

(5) If e is a function reference to a function, f, other than a statement or intrinsic function, the argument sequence of e is $\langle x_1 \ldots x_n \rangle$, and subn is the element of c whose name is f and whose arguments are $\langle a_1 \ldots a_n \rangle$, then the output assumption for e is the conjunction of the output assumptions for $x_1, \ldots, x_n$, conjoined with the result of applying the following term substitution to the output assertion of subn:

term	to be replaced with
STATE	stp
(a_1 STATE)	[x_1 ,stp]
...	
(a_n STATE)	[x_n, stp]
ANS	[e, stp]

Example. Suppose SUM is a function subprogram of two arguments, A and MAX, that SUM has no global names, and that the output assertion of

SUM is (EQUAL ANS (SIGMA 1 (MAX STATE) (A STATE))). Suppose that I and J are local INTEGER names in sub and that VCT is an array in COMMON block BLK in sub. Then the statification, with respect to (BEGIN), of the FORTRAN expression (I+SUM(VCT, J)) is

```
(ZPLUS (I (BEGIN))
       (SUM (BLK-VCT (BEGIN)) (J (BEGIN)))),
```

and the output assumption, with respect to (BEGIN), for (I+SUM(VCT, J)) is

```
(EQUAL (SUM (BLK-VCT (BEGIN)) (J (BEGIN)))
       (SIGMA 1 (J (BEGIN)) (BLK-VCT (BEGIN)))).
```

In particular, the output assumption for an expression tells us what we may assume about the values returned by the user-defined function subprograms in the expression.

Definition. conjoined output assumptions. If $\langle x_1 \ldots x_n \rangle$ is a sequence of expressions and stp is a state term, then the conjoined output assumptions for $\langle x_1 \ldots x_n \rangle$ with respect to stp is the formula obtained by conjoining the output assumption (with respect to stp) for each x_i in $\langle x_1 \ldots x_n \rangle$.

Note. The next two definitions introduce the notions of the "definedness condition" for an expression and the "input condition" for an expression. Intuitively, the input condition for an expression must hold when the expression is evaluated by the processor. For example, the input condition for the reference of a function subprogram is an instantiation of its input assertion. Built-in operations, such as "/", also have input conditions. For example, part of the input condition for (X/Y) is that the value of Y be non-0.

There are two subtleties to the definition of input condition: the first is our recognition of the finite precision of arithmetic; the second is our enforcement of the rule that certain entities must be defined in order to be used.

Part of the input condition for the expression (X+Y) is that the sum of the values of X and Y be expressible on the machine. The reader ought to ask: "Do they know that X and Y are expressible?" The answer is yes. We require that every constant mentioned in an expression be expressible and that every built-in operation produce expressible results.

FORTRAN requires that certain entities be defined when they are used in certain ways. For example, e_1 and e_2 must both be defined in (e_1+e_2). One's first impression is that we should require that all expressions (except array references) produce defined results. Since all function subprograms will be proved to return defined results, and since the built-in functions and operations return defined results, it is only necessary to ensure that variable and array element references are defined. We could therefore define the "input condition" for variable and array element references to require the definedness of the resulting value.

However, one's first impression is often incorrect down in the nitty gritty. We must not require that all variable references produce defined results. The classic example is the use of such an expression in a subroutine argument position that is smashed before it is first referenced -- as happens when one is using the argument position to return results from the subroutine.

Therefore, the treatment of the definedness issue in the definition of "input condition" is not quite what one might expect. Instead of simply making all variable references have a definedness requirement as their input condition and letting (e_1+e_2) inherit that requirement naturally from its subexpressions, we make the input condition for variable references be (TRUE), and we make each composite expression explicitly require the definedness of its immediate variable subexpressions.

Definition. definedness condition. The <u>definedness condition for</u> an expression e <u>with respect to</u> a state term stp is defined as follows:

(1) If e is a variable reference or array element reference, the definedness condition for e is (DEFINEDP [e, stp]).

(2) Otherwise, the definedness condition for e is (TRUE).

Definition. input condition. We now define inductively the concept of the input condition for an expression e with respect to a state term stp. Throughout the definition, all input conditions, definedness conditions, and output assumptions are understood to be with respect to stp.

(1) If e is a constant of type t, the input condition for e is (TRUE) if t is LOGICAL and otherwise is

```
(EXPRESSIBLE.uNUMBERP e)
```

where u is Z, R, D, or c according as t is INTEGER, REAL, DOUBLE, or COMPLEX.

(2) If e is a token or a variable or array reference, the input condition for e is (TRUE).

(3) If e is an array element reference to a with subscript expressions $\langle x_1 \ldots x_n \rangle$, $\langle a\ t\ d_1 \ldots d_n \rangle$ is the member of the array patterns of s with name a, and hyp is the conjoined output assumptions for $\langle x_1 \ldots x_n \rangle$, then the input condition for e is the conjunction of the following

(a) the conjunction of the input condition for each x_i in $\langle x_1 \ldots x_n \rangle$, and

(b) the conjunction, for i from 1 to n, of

```
(IMPLIES hyp
         (AND (ZLESSEQP 1 [x_i, stp])
              (ZLESSEQP [x_i, stp] [d_i, stp])))
```

(which asserts that each array subscript is within the appropriate bounds).

(4) If e is an arithmetic, relational, or logical expression, or e is a function reference to an intrinsic function, the argument sequence of e is $\langle x_1 \ldots x_n \rangle$, and hyp is the conjoined output assumptions for $\langle x_1 \ldots x_n \rangle$, then the input condition for e is the conjunction of the following:

(a) the conjunction of the input condition for each x_i in $\langle x_1 \ldots x_n \rangle$,

(b) the conjunction of the definedness condition for each x_i in $\langle x_1 \ldots x_n \rangle$, and

(c) the implication from hyp to the result of applying the term substitution that replaces (I STATE) by $[x_1, stp]$ and (J STATE) by $[x_2, stp]$ in the "input condition formula" given in Appendix B for the function symbol of the term [e].

(5) For all f and for all expressions $x_1, \ldots, x_n$, if e has the form

$$f(x_1, \ldots, x_n)$$

and f is the name of a statement function pattern of s, and hyp is the conjoined output assumptions for $\langle x_1 \ldots x_n \rangle$, then the input condition for e is the conjunction of:

(a) the conjunction of the input condition for each x_i in $\langle x_1 \ldots x_n \rangle$,

(b) the conjunction of the definedness condition for each x_i in $\langle x_1 \ldots x_n \rangle$, and

(c) the input condition for the expression that results from substituting into the body of the definition of f the expressions $x_1, \ldots, x_n$ for the corresponding arguments of the definition of f.

(6) For each f, for all expressions $x_1, \ldots, x_n, a_1, \ldots, a_n$, and for each subn, if e has the form

$$f(x_1, \ldots, x_n)$$

and f is the name of a function pattern of s, subn is the member of c whose name is f, $a_1, \ldots, a_n$ are the arguments of subn, and hyp is the conjoined output assumptions for $\langle x_1 \ldots x_n \rangle$, then the input condition for e is the conjunction of each of the following:

(a) the conjunction of the input condition for each x_i in $\langle x_1 \ldots x_n \rangle$,

(b) the conjunction of the definedness condition for each x_i in $\langle x_1 \ldots x_n \rangle$, and

(c) the implication from hyp to the result of applying the following term substitution to the input assertion of subn:

term	to be replaced with
STATE	stp
(a_1 STATE)	[x_1, stp]
...	...
(a_n STATE)	[x_n, stp]

We are now ready to begin considering the paths through the subprogram being verified. Consider an edge e on such a path and imagine we have just executed the statement in the head of the edge. How is the state produced by executing that statement related to the state immediately preceding it? We make this clear by defining the "assumptions" for the edge. When we find ourselves faced with proving the input assertions, say, of the next statement (the one in the tail of the edge), we get to assume the assumptions for the edge and all the preceding edges on the path.

Definition. assumption. For each path p in PATHS and for each edge e in p, we define the assumption for e as follows. For the first edge of p the assumption is the result of replacing STATE with (BEGIN) in the assertion for the head of e. For the other edges e of p, the assumption is defined according to the kind of instruction of the statement, st, of the head of e. Let stp and stn be respectively the state term of the edge preceding e and the state term of e. All output assumptions mentioned are with respect to stp.

(1) For each symbolic name v and for each exp, if st has the form

    v = exp

    then the assumption for e is the conjunction of the output assumption for exp, the equation

    (EQUAL [v, stn] [exp, stp])

    and the equations

```
(EQUAL (u stn) (u stp)),
```

for each of the local and global names u of sub except the long name of v.

(2) For each symbolic name v and for all expressions i and exp, if st has the form

```
v(i) = exp
```

then the assumption for e is the conjunction of the output assumption for i, the output assumption for exp, the formulas

```
(EQUAL (ELT1 [v, stn] [i, stp])
       [exp, stp]),

(IMPLIES (NOT (EQUAL I [i, stp]))
         (EQUAL (ELT1 [v, stn] I)
                (ELT1 [v, stp] I))),
```

and the equations

```
(EQUAL (u stn) (u stp)),
```

for each of the local and global names u of sub except the long name of v.

(3) For each symbolic name v and for all expressions i, j, and exp, if st has the form

```
v(i, j) = exp
```

then the assumption for e is the conjunction of the output assumption for i, the output assumption for j, the output assumption for exp, the formulas

```
(EQUAL (ELT2 [v, stn] [i, stp] [j, stp])
       [exp, stp]),

(IMPLIES (NOT (AND (EQUAL I [i, stp])
                   (EQUAL J [j, stp])))
         (EQUAL (ELT2 [v, stn] I J)
                (ELT2 [v, stp] I J))),
```

and the equations

```
(EQUAL (u stn) (u stp)),
```

for each of the local and global names u of sub except the long name of v.

(4) For each symbolic name v and for all expressions i, j, k, and exp, if st has the form

```
v(i, j, k) = exp
```

then the assumption for e is the conjunction of the output assumption for i, the output assumption for j, the output assumption for k, the output assumption for exp, the formulas

```
(EQUAL (ELT3 [v, stn] [i, stp] [j, stp] [k, stp])
       [exp, stp]),

(IMPLIES (NOT (AND (EQUAL I [i, stp])
                   (EQUAL J [j, stp])
                   (EQUAL K [k, stp])))
         (EQUAL (ELT3 [v, stn] I J K)
                (ELT3 [v, stp] I J K))),
```

and the equations

```
(EQUAL (u stn) (u stp)),
```

for each of the local and global names u of sub except the long name of v.

(5) For each symbolic name v and for each k, if st has the form

```
ASSIGN k TO v
```

then the assumption for e is the conjunction of the equation

```
(EQUAL [v, stn] k)
```

and the equations

```
(EQUAL [u, stn] [u, stp])
```

for each of the local and global names u of sub except v.

(6) For each symbolic name i, for each j, and for all labels k_1, ..., and k_n, if e is the jth edge leading from its head and st has the form

```
GO TO i, (k1, ..., kn)
```

then the assumption for e is (EQUAL [i, stp] k_j).

(7) For each symbolic name i, for each j, and for all labels k_1, ..., and k_n, if e is the jth edge leading from its head and st has the form

GO TO (k_1, ..., k_n), i

then the assumption for e is (EQUAL [i, stp] j).

(8) For each type t, for each u, for each expression exp, and for all labels l_1, l_2, and l_3, if st has the form

IF (exp) l_1, l_2, l_3

and exp is an expression of sort <t>, and u is the letter Z, R, or D according as t is INTEGER, REAL, or DOUBLE, then the assumption for e is the conjunction of the output assumption for exp and the term

(uLESSP [exp, stp] (uZERO)),

(uEQP [exp, stp] (uZERO)),

or

(uGREATERP [exp, stp] (uZERO)),

according to whether e is the first, second, or third edge leading from its head.

(9) For each member subn of c (with name subr), and for all expressions e_1, ..., and e_n, if st has the form:

CALL subr(e_1, ..., e_n)

or

CALL subr

and if the argument names of subn are a_1, ..., a_n, and <b_1 ... b_m> is the subsequence of <a_1 ... a_n> containing just the members of <a_1 ... a_n> that are possibly smashed by subn in c, and <c_1 ... c_m> is the corresponding subsequence of <e_1 ... e_n>, then the assumption for e is the conjunction of each of the following:

(a) the conjoined output assumptions for <e_1 ... e_n> (provided st has the first form above),

(b) the result of applying the following term substitution to the output assertion of subn:

term	to be replaced with
STATE	stp
NEWSTATE	stn
(a_1 STATE)	[e_1, stp]

```
    ...                 ...
(a_n STATE)         [e_n, stp]
(b_1 NEWSTATE)      [c_1, stn]
    ...                 ...
(b_m NEWSTATE)      [c_m, stn]
```

and

(c) each equation eq such that for some k, eq has the form

(EQUAL (k stn) (k stp))

and each of the following is true: (i) k is a local or global name of sub, (ii) if k is a global name of sub, then k is not possibly smashed by subn in c, and (iii) k is not the long name of any member of $\langle c_1 \ldots c_m \rangle$.

(10) For each expression exp and statement stm, if st has the form

IF (exp) stm

then the assumption for e is the conjunction of the output assumption for exp and the term (EQUAL [exp, stp] (TRUE)) or (EQUAL [exp, stp] (FALSE)) according as e is the first or the second edge leading from its head.

(11) If st has any other form (i.e., st is an unconditional GO TO statement, RETURN, CONTINUE, STOP, or PAUSE statement) the assumption for e is (TRUE).

Note. We next define what must be proved when the statement in the tail of an edge is encountered.

Definition. verification condition. For each path p in PATHS and for each edge e in p, we now define the verification condition for e. Let n be the node that is the tail of e. Let st be the statement of n. Let stp be the state term of e.

(1) If e is the last member of p, the verification condition for e is the conjunction of (a) the result of substituting stp for STATE in the assertion for n and (b) (TRUE) if st is a RETURN or STOP statement, and otherwise the lexicographic comparison of (i) the result of substituting stp for STATE in the clock for n with (ii) the result of substituting (BEGIN) for STATE in the clock for the first node of p.

(2) For all expressions v and x, if st has the form

```
v = x
```

then the verification condition for e is the conjunction of the definedness condition for x with respect to stp, the input condition for x with respect to stp, and the input condition for v with respect to stp.

(3) For each symbolic name v and for all labels k_1, ..., and k_n, if st has the form

```
GO TO v, (k1, ..., kn)
```

then the verification condition for e is the disjunction of the terms (EQUAL [v, stp] k_j) for each j from 1 to n.

(4) For each symbolic name v and for all labels k_1, ..., and k_n, if st has the form

```
GO TO (k1, ..., kn), v
```

then the verification condition for e is the formula

```
(AND (ZLESSEQP 1 [v, stp])
     (ZLESSEQP [v, stp] n))
```

(5) For each expression x, for all labels l_1, l_2, and l_3, and for each statement stm, if st has the form

```
IF (x) l1, l2, l3
```

or

```
IF (x) stm
```

then the verification condition for e is the conjunction of the definedness condition for x with respect to stp and the input condition for x with respect to stp.

(6) For each symbolic name subr, for all expressions x_1, ..., and x_n, and for all subn, a_1, ..., a_n, and hyp, if subn is the subroutine subprogram of c with name subr, a_1, ..., a_n are the arguments of subn, hyp is the conjoined output assumptions for $\langle x_1 \ldots x_n \rangle$ with respect to stp, and st has the form

```
CALL subr(x1, ..., xn)
```

or

CALL subr

then the verification condition for e is the conjunction of:

(a) the implication from hyp to the result of applying the following term substitution to the input assertion of subn:

term	to be replaced with
STATE	stp
(a_1 STATE)	[x_1, stp]
...	...
(a_n STATE)	[x_n, stp]

and

(b) the conjunction of the input conditions for each x_i in <x_1 ... x_n>.

(7) If st is a GO TO assignment, unconditional GO TO, CONTINUE, or PAUSE statement, the verification condition for e is (TRUE).

Note. We now combine all the foregoing concepts to define what we mean when we say that a syntactically correct collection of subprograms is "semantically correct" with respect to some input/output specifications and a FORTRAN theory. Semantically correct contexts are constructed incrementally from the empty context by adding a single new subprogram, specifying and annotating it, and then proving it correct by proving each of its verification conditions, under the assumptions governing each verification condition.

Definition. semantically correct. We define recursively the notion that a context is semantically correct with respect to an input/output specification <i o> and a theory. Let phi be the empty function. Recall that we have fixed upon a T, c, sub, s, seq, labs, inpt, outpt, inpclk, lpinv, and lpclk.

(1) The empty context is semantically correct with respect to <phi phi> and T.

(2) Suppose that c′ is semantically correct with respect to <inpt′ outpt′> and T. Suppose further that the restrictions of inpt and outpt to c′ are, respectively,

inpt′ and outpt′, and that c is obtained by adding sub to the end of c′. Then c is semantically correct with respect to <inpt outpt> and T if

(a) every node in the extended flow graph of sub is reachable from 0, and

(b) for each path p in PATHS and for each edge e in p, the verification condition for e is a logical consequence (in the tertiary verification extension of T for c and sub) of the global assumptions and the assumptions for e and for all of the edges that precede e in p.

XI THE DO STATEMENT

We now describe how we handle the FORTRAN DO statement. Our description is informal but complete.

We define the DO statement to be a statement of the form:

```
DO lab v = i, j, k
```

or

```
DO lab v = i, j
```

where lab is a label, v is an INTEGER variable name, and each of i, j, and k is an INTEGER variable name, a positive INTEGER constant, or a token. The second form of DO is an abbreviation for an instance of the first in which k is 1. We will henceforth restrict our attention to the first form.

We extend the definition of "possibly smashed" so that the DO statement:

```
DO lab v = i, j, k
```

possibly smashes the long name of v.

We permit a subprogram <s seq labs> to contain DO statements among the executable statements provided the following additional syntactic constraints are met.

(1) If seq contains a statement, $stmt_1$, of the form:

```
DO lab v = i, j, k
```

then lab must be the statement label of some statement, $stmt_2$, after $stmt_1$ in seq. $stmt_2$ must be an assignment, GO TO assignment, or CONTINUE statement, or a logical IF containing one of the three preceding kinds of

statements. The sequence of statements from and including the first statement after $stmt_1$ through and including $stmt_2$ is called the range of $stmt_1$.

(2) All DOs must be "nested" in the sense that if the ranges of two DO statements are not disjoint, then the range of one contains the range of the other.

(3) No label used in an unconditional, computed, or assigned GO TO, or in an arithmetic or logical IF statement, stmt, may be attached to a statement within the range of a DO not containing stmt.

(4) No statement within the range of a DO statement of the form:

```
DO lab v = i, j, k
```

may possibly smash the long name of any of the variables among v, i, j, and k.

After we have accepted a subprogram as syntactically well formed, we translate the subprogram to an equivalent one that does not contain DO statements. All concepts relating to the "semantic correctness" of the original subprogram are defined in terms of the semantic correctness of the translated program. For example, the flow graph for the original subprogram is defined to be the flow graph for the translation of the subprogram.

To translate a subprogram we replace each DO statement and its range by a new sequence of statements. Let the DO statement and its range be described schematically by:

```
nnn  DO lab v = i, j, k
     stmt1
      .
      .
      .
lab  stmtn
```

For generality, we suppose that the DO statement itself has a statement label nnn, so that we can describe how the statement labels in the new program are related to those in the original program. We replace the above sequence of statements with:

```
nnn  IF (((i.GT.j).OR.((i.LT.1).OR.(k.LT.1)))) STOP
     v = i
top  CONTINUE
     IF ((v.GT.j)) GO TO fin
     stmt1
      .
      .
      .
lab  stmtn
     v = (v + k)
     GO TO top
fin  CALL UNDEFINER(v)
```

The two statement labels top and fin generated for a given DO statement are different from the labels generated for any other DO statement of the subprogram and are different from all labels in the original subprogram. The old statement label nnn is attached to the logical IF statement indicated above. Labels on statements $stmt_1$ through $stmt_n$ in the original subprogram are attached to statements $stmt_1$ through $stmt_n$ in the translated subprogram.

The logical IF at nnn is generated in accordance with the ANSI FORTRAN 66 requirement that the initial values of i, j, and k be such that i is greater than j (so the DO is allowed to cycle at least once), and all be greater than 0. We enforce the ANSI restriction by testing its negation and executing STOP if the ANSI restriction is not met. The path on which the ANSI restriction is assumed not to hold requires that we prove (FALSE), the assertion for STOP. Thus, we must be able to establish that the ANSI restriction holds at nnn.

ANSI requires that the variables among i, j, and k must be defined upon the execution of the DO statement. In addition, the constants among i, j, and k must be expressible integers. Both these requirements are enforced by the normal processing of the logical expression in the logical IF at nnn.

The CONTINUE statement labeled top is provided as a node which may be annotated with an assertion.*

At the statement labeled fin we CALL the subroutine (named, in this document) UNDEFINER, giving it the DO statement's control variable, v. UNDEFINER is built into the verification condition generator and is known to "possibly smash" its argument and have no affect on any other name. Both the input and output assertion for UNDEFINER are (TRUE). Thus, in accordance with ANSI FORTRAN 66, the value of v upon the normal completion of the DO loop is unknown. This clause in the ANSI specification permits different compilers to implement the loop control differently.

---

* We do not spell out in this document the precise means we provide for the user to annotate his program. However, we provide a mechanism by which the user can write a loop invariant for the DO, and that invariant is attached to the CONTINUE statement at top.

XII USING SEMANTICALLY CORRECT CONTEXTS

We have formally defined what it means for a FORTRAN subprogram to be semantically correct with respect to an input/output specification and a theory. We have not related our formal notions to the real world of FORTRAN computing. In particular, if the FORTRAN programmer has a semantically correct context (e.g., a "library" of "correct" subroutines), how does he use it subprograms and what do they do?

It would be necessary to formalize all of FORTRAN to answer this question with the same level of precision and formality with which we have defined "semantic correctness." However, we can describe vaguely how semantically correct contexts may be used. Because of its vagueness, the following description must be taken with a grain of salt.

To obtain a useable FORTRAN program from a context c that is semantically correct with respect to some input and output specification and a theory T, the following steps are performed:

(1) The theory T is extended so as to specify the values of certain undefined functions. In particular, for each token, token, an axiom is added equating (token) with some expressible positive integer. In addition, axioms are added that equate (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER) and (GREATEST.INEXPRESSIBLE.NEGATIVE.INTEGER) with integers satisfying the axiom INTEGER.SIZE of Appendix A.

(2) The user then checks that, in the light of the extended theory, the input and output specifications suit his needs. (For example, given the newly specified values of the tokens, do the input specifications permit the desired applications of the subprograms?)

(3) The user checks that the FORTRAN processor on which he intends to operate executes the built-in arithmetic, logical, and intrinsic functions in accordance with the definitions of Appendix A and the input conditions of Appendix B. (For example, does the processor's integer addition mechanism really return the mathematical sum of two integers whenever that sum is in the interval between

(LEAST.INEXPRESSIBLE.POSITIVE.INTEGER) and
(GREATEST.INEXPRESSIBLE.NEGATIVE.INTEGER)?)

(4) All occurrences of tokens in the subprograms of c are replaced by their axiomatized values and the subprograms are printed according to the standard FORTRAN rules regarding lines and columns.

(5) The printed subprograms are then combined with unverified subprograms and a main program. The main program must include a COMMON declaration of each COMMON block of c. Each such COMMON block must be declared in the main program program precisely as it is declared in any member of c (including type and dimension information).

(6) The user checks that the combined executable program does not exceed the "capacity" of his processor (see section 1.2.1 of [12] and section 1.3.2 of [1]). This will include checking that there is enough "room" to store the arrays and subprograms. It could conceivably involve more bizarre interpretations of "capacity" such as restrictions on the number of formal arguments or subroutine CALLs. What is actually required here is a formal characterization of the "capacity" of the processor in question.

Subprograms in the verified context may be called from unverified programs and are guaranteed to terminate without run-time error and produce results consistent with their input/output specifications. Of course, such calls must satisfy the FORTRAN rules for invoking subprograms (e.g., the types of the actuals must correspond to the types of the formals, adjustable array dimensions must be property defined, and so on).

XIII AN EXAMPLE

In a 1977 Communications of the ACM article [4], we described an algorithm for finding the first occurrence of one character string, PAT, in another, STR. The algorithm is currently the fastest known way to solve this problem on the average. Our algorithm has two unusual properties. First, in verifying that PAT does not occur within the first i characters of STR the algorithm will typically fetch and look at fewer than i characters. Second, as PAT gets longer the algorithm speeds up. That is, the algorithm typically spends less time to find long patterns than short ones.

In Chapter XVIII of [5] we present a version of the algorithm coded in a simple "toy" programming language that -- like many languages used in program verification -- ignores many issues raised by conventional programming languages. In this section we discuss the verification of the same version of the algorithm, but this time coded in our FORTRAN subset. Our subroutine finds the first occurrence of one array of "character codes" in another array of "character codes." By "character codes" we mean INTEGERs in the range 1 to @ASIZE, a token understood to be the size of the alphabet (e.g., 128 for ASCII).

String searching is not FORTRAN's forte. However, we chose this example for four reasons. First, the algorithm is of interest both theoretically and practically and is in day-to-day use in certain text processing systems. Second, the algorithm has been published, illustrated, and carefully explained elsewhere. Third, the algorithm presents certain interesting features from the point of view of verification. Finally, it is interesting to contrast the verification of a "toy" version with the verification of exactly the same algorithm in a real language.

A. The Implementation in FORTRAN

The whole idea behind the algorithm is illustrated by the following example. Suppose we are trying to find PAT in STR and, having scanned some initial part of STR and failed to find PAT, are now ready to ask whether PAT occurs at the position marked by the arrow below:

```
PAT:                  EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE
                      ^
```

Instead of focusing on the left-hand end of the pattern (i.e., on the "E" indicated by the arrow) the algorithm considers the right-hand end of the pattern. In particular, the algorithm fetches the "I" in the word "SIMPLE." Since "I" does not occur in PAT, the algorithm can slide the pattern down by seven (the length of PAT) without missing a possible match. Afterwards, it focuses on the end of the pattern again, as marked by the arrow below.

```
PAT:                        EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE
                            ^
```

In general, as the next step would suggest, the algorithm slides PAT down by the number of characters that separate the end of the pattern from the last occurrence in PAT of the character, c, just fetched from STR (or the length of PAT if c does not occur in PAT). In the configuration above, PAT would be moved forward by five characters, so as to align the "X" in PAT with the just fetched "X" in STR.

If the algorithm finds that the character just fetched from STR matches the corresponding character of PAT, it moves the arrow backwards and repeats the process until it either finds a mismatch and can slide PAT forward, or matches all the characters of PAT.

The algorithm must be able to determine efficiently for any character c, the distance from the last occurrence of c in PAT to the right-hand end of PAT. But since there are only a finite number of

characters in the alphabet we can preprocess PAT and set up a table that answers this question in a single array access.

The reader is referred to [4] for a thorough description of an improved version of the algorithm that can be implemented so as to search for PAT through i characters of STR and execute less than i machine instructions, on the average. In addition, [4] contains a statistical analysis of the average case behavior of the algorithm and discusses several implementation questions.

A FORTRAN version of the algorithm is exhibited below. The subroutine FSRCH is the search algorithm itself; it takes five arguments, PAT, STR, PATLEN, STRLEN, and X. PAT and STR are one-dimensional adjustable arrays of length PATLEN and STRLEN respectively. X is the dummy argument into which the answer is smashed. The answer is either the index into STR at which the winning match is found, or else it is STRLEN+1 indicating no match exists.

FSRCH starts by CALLing the subroutine SETUP, which preprocesses PAT and smashes the COMMON array DELTA1. DELTA1 has one entry for each character code in the alphabet. SETUP executes in time linear in PATLEN. It initializes DELTA1 as though no character occurred in PAT and then sweeps PAT once, from left to right, filling in the correct value of DELTA1 for each character occurrence, as though that occurrence were the last occurrence of the character in PAT. Thus, if the same character occurs several times in PAT (as "E" does in "EXAMPLE") then its DELTA1 entry is smashed several times and the last value is the correct one.

```
      SUBROUTINE FSRCH(PAT, STR, PATLEN, STRLEN, X)
      INTEGER DELTA1
      INTEGER PATLEN
      INTEGER STRLEN
      INTEGER PAT
      INTEGER STR
      INTEGER I
      INTEGER J
      INTEGER C
      INTEGER NEXTI
      INTEGER X
      INTEGER MAX0
      DIMENSION DELTA1(@ASIZE)
      DIMENSION PAT(PATLEN)
      DIMENSION STR(STRLEN)
      COMMON /BLK/DELTA1
      CALL SETUP(PAT, PATLEN)
      I = PATLEN
200   CONTINUE
      IF ((I.GT.STRLEN)) GO TO 500
      J = PATLEN
      NEXTI = (1+I)
300   CONTINUE
      C = STR(I)
      IF ((C.NE.PAT(J))) GO TO 400
      IF ((J.EQ.1)) GO TO 600
      J = (J-1)
      I = (I-1)
      GO TO 300
400   I = MAX0((I+DELTA1(C)), NEXTI)
      GO TO 200
500   X = (STRLEN+1)
      RETURN
600   X = I
      RETURN
      END
```

```
      SUBROUTINE SETUP(A, MAX)
      INTEGER DELTA1
      INTEGER A
      INTEGER MAX
      INTEGER I
      INTEGER C
      DIMENSION DELTA1(@ASIZE)
      DIMENSION A(MAX)
      COMMON /BLK/DELTA1
      DO 50 I=1, @ASIZE
      DELTA1(I) = MAX
50    CONTINUE
      DO 100 I=1, MAX
      C = A(I)
      DELTA1(C) = (MAX-I)
100   CONTINUE
      RETURN
      END
```

As described in this document, our subset allows only one variable to be declared in each INTEGER statement, requires the declaration of implicitly typed INTEGER variables such as I and J, and requires full parenthesization of expressions. These restrictions could be relaxed somewhat. The statements labeled 200 and 300 in FSRCH are CONTINUE statements to permit the attachment of loop assertions at those points.

Chapter XVIII of [5] discusses the "toy" version of the algorithm implemented above and fully illustrates the algorithm at work. We highly recommend that the reader see Chapter XVIII before continuing with this discussion. In particular, that chapter contains a description of the algorithm in which we devote a paragraph to virtually every statement in the code for FSRCH. Furthermore, we carefully derive the input/output assertions for the algorithm, discuss, from an intuitive point of view, the invariants that are being maintained, express those invariants formally, derive the verification conditions (ignoring such things as aliasing, overflow, and array bounds violations), and prove the verification conditions. In the following discussion we assume the reader understands how the algorithm works.

B. The FORTRAN Theory

To specify the input and output assertions for these two subroutines we must extend the basic FORTRAN theory by the introduction of the mathematical concepts of (a) a sequence being a "character string" on a given sized alphabet, (b) the initial segments of two strings "matching," (c) the leftmost match of PAT in STR, and (d) the distance from the last occurrence of C in PAT to the end of PAT. Below we give the definitions of these mathematical functions.

```
Definition.
(STRINGP A I SIZE)
   =
(IF (ZEROP I)
    T
    (AND (NUMBERP (ELT1 A I))
         (NOT (EQUAL (ELT1 A I) 0))
         (NOT (LESSP SIZE (ELT1 A I)))
         (STRINGP A (SUB1 I) SIZE)))

Definition.
(MATCH PAT J PATLEN STR I STRLEN)
   =
(IF (LESSP PATLEN J)
    T
    (IF (LESSP STRLEN I)
        F
        (AND (EQUAL (ELT1 PAT J) (ELT1 STR I))
             (MATCH PAT
                    (ADD1 J)
                    PATLEN STR
                    (ADD1 I)
                    STRLEN))))

Definition.
(SEARCH PAT STR PATLEN STRLEN I)
   =
(IF (LESSP STRLEN I)
    (ADD1 STRLEN)
    (IF (MATCH PAT 1 PATLEN STR I STRLEN)
        I
        (SEARCH PAT STR PATLEN STRLEN
                (ADD1 I)))))
```

```
Definition.
(DELTA1 A C MAX)
   =
(IF (ZEROP MAX)
    0
    (IF (EQUAL C (ELT1 A MAX))
        0
        (ADD1 (DELTA1 A C (SUB1 MAX)))))
```

For example, (MATCH PAT J PATLEN STR I STRLEN) determines whether the characters of PAT in positions J through PATLEN are equal to the corresponding characters of STR starting at position I and not exceeding STRLEN. MATCH is recursive. That is, provided J $\leq$ PATLEN and I $\leq$ STRLEN, MATCH checks that the J^{th} character of PAT is equal to the I^{th} character of STR and, if so, requires that there be a MATCH starting at positions I+1 and J+1. The recursive function SEARCH is the mathematical expression of the naive string searching algorithm. (SEARCH PAT STR PATLEN STRLEN I) asks, for each position in STR between I and STRLEN, whether a MATCH with PAT occurs at that position.

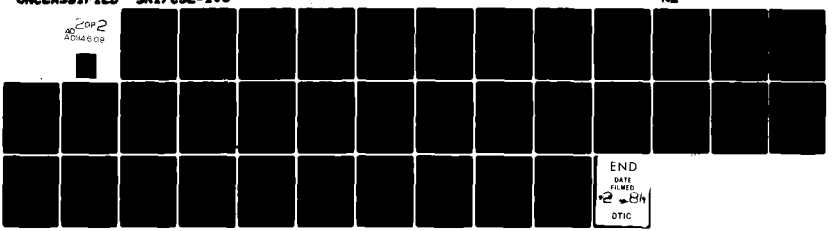
We extend the basic FORTRAN theory by adding the definitions above. The result is a FORTRAN theory theory appropriate for a context containing SETUP and FSRCH.

C. The Specification of SETUP

To verify SETUP we must first specify it with input and output assertions. These assertions must be expressed in terms of the formal arguments to SETUP and its global names. Note that our implementation of SETUP used the formal arguments A and MAX. In the actual CALL of SETUP from FSRCH A will be PAT and MAX will be PATLEN. We chose different names to make instantiations of the input and output assertions more obvious.

The input assertion for SETUP is:

AD-A094 609 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 12/1
A VERIFICATION CONDITION GENERATOR FOR FORTRAN.(U)
JUN 80 R S BOYER, J S MOORE N00014-75-C-0816
UNCLASSIFIED SRI/CSL-103 NL



```
(AND (STRINGP (A STATE) (MAX STATE) (@ASIZE))
     (NOT (EQUAL (MAX STATE) 0))
     (NUMBERP (MAX STATE))
     (LESSP (ADD1 (@ASIZE))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER))
     (LESSP (ADD1 (MAX STATE))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER)))
```

The assertion requires that the elements of A from 1 to MAX be character codes in the alphabet of size @ASIZE, and that MAX be a positive INTEGER. In addition, it requires that both @ASIZE+1 and MAX+1 be expressible.

The reader may ask "Why the +1s? Why not simply require that @ASIZE and MAX be expressible?" An inspection of the ANSI FORTRAN 66 semantics for DO-loops reveals that

```
DO 100 I=1, MAX
```

causes I to be set to MAX+1 immediately before the termination condition is checked for the last time. Thus, unless MAX+1 is expressible, the last increment will either cause an overflow error or cause I to be set to garbage.

The reader may also ask "Why do you prohibit a MAX of 0? Doesn't that just correspond to the empty string?" One's first considered reaction might be that the condition is present because MAX is the dimension of an array declared in SETUP. However, that aspect of MAX's use does not show up in the verification conditions generated for SETUP; So why is it in the input assertion for SETUP? The answer is because of the

```
DO 100 I=1, MAX
```

statement. FORTRAN 66 requires that on entry to the DO loop the maximum, i.e., MAX, be greater than or equal to the initial value, 1. Should MAX ever be 0, a program with the above DO statement in it will cause unpredictable behavior on some correct FORTRAN 66 processors.

The output assertion for SETUP is:

```
(IMPLIES (AND (NUMBERP C)
              (NOT (EQUAL C 0))
              (NOT (LESSP (@ASIZE) C)))
         (EQUAL (ELT1 (BLK-DELTA1 NEWSTATE) C)
                (DELTA1 (A STATE)
                        C
                        (MAX STATE))))
```

The assertion relates NEWSTATE (the state at the conclusion of the execution of SETUP) to STATE (the state at the beginning of the execution). Informally, it says that for every character C in the alphabet, the Cth element of the DELTA1 array in COMMON block BLK (at the conclusion of the execution) is equal to the distance from the last occurrence of C in A to the end of A, where that distance is defined by the mathematical function DELTA1.

Note that, as required of an output assertion, the term above is a term in the primary verification extension of our theory. That is, it is well-formed and mentions no function symbols other than those in our extension of the basic FORTRAN theory and the arguments and global names of SETUP. Note also that it incarcerates STATE and NEWSTATE: they appear only as arguments to the arguments and globals of SETUP.

The system has proved SETUP correct as specified above. To describe the verification of SETUP we would have to annotate it with an input clock, loop invariants, and loop clocks for the two DO loops. Since we go into the verification of FSRCH in some detail, we will now skip over the details of SETUP. Assume SETUP has been proved and that we now have a semantically correct context that contains it.

Let us jump ahead for a moment and consider the following questions. What do we have to prove when we encounter

```
CALL SETUP(PAT, PATLEN)
```

in another subprogram (e.g., FSRCH)? Suppose the state term before the CALL is (START). Then we must prove the verification condition generated for the CALL, which is the conjunction of:

(a) a term substitution instance of the input assertion for SETUP:

```
(AND (STRINGP (PAT (START))
             (PATLEN (START))
             (@ASIZE))
     (NOT (EQUAL (PATLEN (START)) 0))
     (NUMBERP (PATLEN (START)))
     (LESSP (ADD1 (@ASIZE))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER))
     (LESSP (ADD1 (PATLEN (START)))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER))), and
```

(b) the conjunction of the input conditions for the actual expressions, which is (TRUE) since they are variables.

On the other side, what do we get to assume after the CALL of SETUP? The new state term is (NEXT (START)). The assumption arising from the CALL is the conjunction of the following:

(a) the conjoined output assumptions for the actual expressions, which is (TRUE) in this case,

(b) a term substitution instance of the output assertion of SETUP:

```
(IMPLIES (AND (NUMBERP C)
              (NOT (EQUAL C 0))
              (NOT (LESSP (@ASIZE) C)))
         (EQUAL (ELT1 (BLK-DELTA1 (NEXT (START)))
                      C)
                (DELTA1 (PAT (START))
                        C
                        (PATLEN (START)))))
```

which tells us that the elements of DELTA1 in the new state are correctly set,

(c) the assumption that PAT and PATLEN (and all the other local and global names of the calling subprogram not possibly smashed by the CALL of SETUP) are unchanged in the new state:

```
(EQUAL (PAT (NEXT (START)))
       (PAT (START)))

(EQUAL (PATLEN (NEXT (START)))
       (PATLEN (START))).
```

In short, in the state after the CALL, DELTA1 is correctly set and no other names have been affected.

D. The Specification of FSRCH

Assuming we have a semantically correct context containing SETUP we now proceed to specify FSRCH. The input assertion for FSRCH is:

```
(AND (LESSP (ADD1 (@ASIZE))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER))
     (STRINGP (PAT STATE) (PATLEN STATE) (@ASIZE))
     (NUMBERP (PATLEN STATE))
     (LESSP 0 (PATLEN STATE))
     (STRINGP (STR STATE) (STRLEN STATE) (@ASIZE))
     (NUMBERP (STRLEN STATE))
     (LESSP 0 (STRLEN STATE))
     (LESSP (PLUS (PATLEN STATE) (STRLEN STATE))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER)))
```

Informally, the assertion puts the same restrictions on @ASIZE and PAT as required by our use of SETUP. In addition, it requires that STR be a nonempty character string on the alphabet of size @ASIZE. The most interesting requirement however is the last: the sum of PATLEN and STRLEN must be expressible.

At first sight one might think that FSRCH will work for any sized PAT and STR as long as every character in them can be indexed. Were that true, it would be enough to require that both PATLEN and STRLEN be expressible. But suppose that PAT has been pushed down STR so that the last character of PAT is aligned with the last character of STR. That is, I is set to STRLEN. Suppose that the last character, C, of STR does not occur in PAT. Then we increment I by the contents of the DELTA1 array at C. That value will be PATLEN, since C does not occur in PAT. Thus, I becomes STRLEN+PATLEN. FSRCH then jumps to 200, discovers that I exceeds STRLEN, and quits. But if STRLEN+PATLEN were not expressible, the step in which we increment I the last time would either cause an overflow error or return garbage.

The output assertion for FSRCH is:

```
(EQUAL (X NEWSTATE)
       (SEARCH (PAT STATE)
               (STR STATE)
               (PATLEN STATE)
               (STRLEN STATE)
               1))
```

Informally, the assertion says that at the conclusion of FSRCH, X is set to the correct value, as defined by applying the mathematical function SEARCH to the initial values of the arguments.

E. The Annotation of FSRCH

To prove FSRCH correct we must annotate it. We first specify the input clock. Recall that a clock is an n-tuple of natural numbers. The "time" remaining on a clock when it is encountered is supposed to be lexicographically smaller than the "time" remaining on the previously encountered clock. The input clock is attached to the entry and thus puts a limit on the total amount of "time" the program can run, expressed as a function of the initial environment.

By inspecting FSRCH we see that every time we go around the loop through statement label 200, I is bigger than it was before -- though this observation requires some understanding of NEXTI and MAX0. I cannot get bigger than STRLEN+PATLEN. So the program can cycle through statement 200 only a finite number of times. However, there is an inner loop, through statement 300. Every time the program cycles through it, J gets smaller and is bounded below by 0. Thus, we have a lexicographic argument that the program terminates. The argument uses clocks with two components. Intuitively, the first component must tick down at least once every time we go through the outer loop. We do not care what the second component does when the first component ticks down. The second component must tick down every time we go through the inner loop -- and when that happens the first component must not increase! Here is the input clock for FSRCH.

```
(LIST (PLUS 1
            (STRLEN (START))
            (PATLEN (START)))
      0)
```

By making the first component large enough in the input clock we do not have to worry about the initial value of the second component.

We now specify the loop invariants and loop clocks for FSRCH. It is at this point that the reader benefits the most from the presentation in Chapter XVIII of [5] because there we explain the role of many of the conjuncts in the two loop invariants. Here is the loop invariant that we attach to the CONTINUE statement at line 200 in FSRCH:

```
(AND (EQUAL (PAT STATE) (PAT (START)))
     (EQUAL (STR STATE) (STR (START)))
     (EQUAL (PATLEN STATE)
            (PATLEN (START)))
     (EQUAL (STRLEN STATE)
            (STRLEN (START)))
     (NUMBERP (I STATE))
     (NOT (LESSP (I STATE) (PATLEN (START))))
     (LESSP (I STATE)
            (PLUS (PATLEN (START))
                  (SEARCH (PAT (START))
                          (STR (START))
                          (PATLEN (START))
                          (STRLEN (START))
                          1)))
     (IMPLIES (AND (NUMBERP C)
                   (NOT (EQUAL C 0))
                   (NOT (LESSP (@ASIZE) C)))
              (EQUAL (ELT1 (BLK-DELTA1 STATE) C)
                     (DELTA1 (PAT (START))
                             C
                             (PATLEN (START))))))
```

The first seven conjuncts are described in [5] (modulo the translation from indices that start at 0 to indices that start at 1). Intuitively, they say that PAT, STR, PATLEN, and STRLEN are not being modified (i.e., they are the same objects in the current state, STATE, as in the initial state, (START)), that there are at least PATLEN

characters to the left of I (so we may compare then pairwise), and that we have not yet passed the right-hand end of the winning match of PAT in STR. The final conjunct says that the DELTA1 array is not being modified. In particular, it says that its configuration in the current state STATE has the same property that it did immediately after we called SETUP.

Here is the loop clock attached to statement 200:

```
(LIST (DIFFERENCE (PLUS 1
                        (STRLEN (START))
                        (PATLEN (START)))
                  (I STATE))
      (ADD1 (PATLEN (START))))
```

Since I is never 0, the first component of this clock is less than that of the input clock. We will have to prove that this clock is bigger than any clock we encounter on a nonexit path leading out of statement 200 (which we can do because the only such path leads to the inner loop where we will hold the first component fixed and count the second down). We will also have to prove that this clock is smaller than any clock we see at the beginning of a path coming into statement 200 (which we can do since, except for the input path, the only such path will come from the inner loop where I will have been increased with the MAX0 expression).

The loop invariant to be attached to the CONTINUE statement at statement 300 in FSRCH is:

```
(AND (EQUAL (PAT STATE) (PAT (START)))
     (EQUAL (STR STATE) (STR (START)))
     (EQUAL (PATLEN STATE)
            (PATLEN (START)))
     (EQUAL (STRLEN STATE)
            (STRLEN (START)))
     (NOT (LESSP (NEXTI STATE)
                 (ADD1 (PATLEN (START)))))
     (LESSP (NEXTI STATE)
            (ADD1 (PLUS (PATLEN (START))
                        (SEARCH (PAT (START))
                                (STR (START))
                                (PATLEN (START))
                                (STRLEN (START))
                                1))))
     (IMPLIES (AND (NUMBERP C)
                   (NOT (EQUAL C 0))
                   (NOT (LESSP (@ASIZE) C)))
              (EQUAL (ELT1 (BLK-DELTA1 STATE) C)
                     (DELTA1 (PAT (START))
                             C
                             (PATLEN (START)))))
     (NUMBERP (I STATE))
     (NOT (EQUAL (I STATE) 0))
     (NUMBERP (J STATE))
     (NOT (EQUAL (J STATE) 0))
     (NUMBERP (NEXTI STATE))
     (NOT (LESSP (PATLEN (START)) (J STATE)))
     (NOT (LESSP (STRLEN (START)) (I STATE)))
     (EQUAL (NEXTI STATE)
            (PLUS (ADD1 (PATLEN (START)))
                  (DIFFERENCE (I STATE) (J STATE))))
     (NOT (LESSP (ADD1 (STRLEN (START)))
                 (NEXTI STATE)))
     (NOT (LESSP (I STATE) (J STATE)))
     (MATCH (PAT (START))
            (ADD1 (J STATE))
            (PATLEN (START))
            (STR (START))
            (ADD1 (I STATE))
            (STRLEN (START)))
     (NUMBERP (ELT1 (BLK-DELTA1 STATE)
                    (ELT1 (STR (START)) (I STATE))))
     (NUMBERP (ELT1 (STR (START)) (I STATE)))
     (NUMBERP (ELT1 (PAT (START)) (J STATE))))
```

The first seven conjuncts are really just a version of the

invariant at 200, which we must maintain because we will have to prove it when we exit the inner loop and jump to 200. The next eleven conjuncts specify the invariant inherently maintained by the inner loop. This invariant is discussed in [5]. Intuitively, it requires that J and I be "corresponding" legal indices into PAT and STR and that we have established that the terminal substrings of PAT and STR starting at J+1 and I+1 MATCH. The last three conjuncts are unnecessary but make the verification a little easier. They inform us that the elements of DELTA1, STR, and PAT that we will access in any single iteration through the inner loop are nonnegative INTEGERs. These facts can be derived. However, by making them explicit we permit the theorem-prover to simplify certain arithmetic expressions more rapidly because it can immediately rule out the possibilities that negative quantities are involved.

The loop clock at statement 300 is:

```
(LIST (DIFFERENCE (PLUS 1
                        (STRLEN (START))
                        (PATLEN (START)))
                  (SUB1 (NEXTI STATE)))
      (J STATE))
```

When we come into the inner loop from the outer, this clock is less than the clock at 200 because the first component is equal and the second component is smaller. Every time we go around the inner loop this clock will be less than it was the previous time because the first component will not have changed and the second will have been decremented by 1. When we go from the inner loop back out to the outer one, the clock at 200 will be smaller because I will be larger than (SUB1 (NEXTI STATE)).

F. The Verification of One Path Through FSRCH

The global assumption for the verification of FSRCH is the conjunction of (1) the partially instantiated input assertion for FSRCH:

```
(AND (LESSP (ADD1 (@ASIZE))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER))
     (STRINGP (PAT (START))
              (PATLEN (START))
              (@ASIZE))
     (NUMBERP (PATLEN (START)))
     (LESSP 0 (PATLEN (START)))
     (STRINGP (STR (START))
              (STRLEN (START))
              (@ASIZE))
     (NUMBERP (STRLEN (START)))
     (LESSP 0 (STRLEN (START)))
     (LESSP (PLUS (PATLEN (START))
                  (STRLEN (START)))
            (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER))
     (DEFINEDP (STRLEN (START)))
     (DEFINEDP (PATLEN (START))))
```

and (2) the axiom defining the types of the variables in FSRCH, when they are defined:

```
(AND (IMPLIES (DEFINEDP (PATLEN STATE))
              (ZNUMBERP (PATLEN STATE)))
     (IMPLIES (DEFINEDP (J STATE))
              (ZNUMBERP (J STATE)))
     (IMPLIES (DEFINEDP (NEXTI STATE))
              (ZNUMBERP (NEXTI STATE)))
     (IMPLIES (DEFINEDP (C STATE))
              (ZNUMBERP (C STATE)))
     (IMPLIES (DEFINEDP (STRLEN STATE))
              (ZNUMBERP (STRLEN STATE)))
     (IMPLIES (DEFINEDP (X STATE))
              (ZNUMBERP (X STATE)))
     (IMPLIES (DEFINEDP (I STATE))
              (ZNUMBERP (I STATE)))
     (IMPLIES (DEFINEDP (ELT1 (PAT STATE)
                              I))
              (ZNUMBERP (ELT1 (PAT STATE)
                              I)))
     (IMPLIES (DEFINEDP (ELT1 (STR STATE)
                              I))
              (ZNUMBERP (ELT1 (STR STATE)
                              I)))
     (IMPLIES (DEFINEDP (ELT1 (BLK-DELTA1 STATE)
                              I))
              (ZNUMBERP
               (ELT1 (BLK-DELTA1 STATE)
                     I)))).
```

We now consider the verification conditions along the path from the input to statement 200. The path contains the artificially added node 0, the call of SETUP, the initialization of I to PATLEN, and the annotated CONTINUE statement labeled 200.

The initial state term is (START). Consider the first edge, which terminates at the CALL statement:

```
CALL SETUP(PAT, PATLEN)
```

We have already exhibited the verification condition for this CALL statement. In particular, we have to prove that the instantiated input assertion for SETUP is satisfied. We get to assume the global assumptions above. The proof is trivial.

The state term after the CALL is (NEXT (START)). We also get to assume the previously discussed assumptions resulting from the CALL of SETUP. Those assumptions relate variables in the (START) state to those in (NEXT (START)). Let us continue down the input path to:

```
I = PATLEN
```

The verification condition for the edge leading to this statement is the conjunction of the input condition for I (which is (TRUE)), the input condition for PATLEN (which is (TRUE)), and the definedness condition for PATLEN in the current state (NEXT (START)). That is, we have to prove:

```
(DEFINEDP (PATLEN (NEXT (START))))
```

assuming the global assumptions and the assumptions resulting from the CALL of SETUP. Those latter assumptions, recall, tell us that (PATLEN (NEXT (START))) is equal to (PATLEN (START)). But the global assumption tells us (DEFINEDP (PATLEN (START))). Thus, this verification condition is trivial also.

As a result of the assignment statement above, the new state term is (NEXT (NEXT (START))). The assumption for the edge coming out of the assignment tells us that (I (NEXT (NEXT (START)))) is (PATLEN (NEXT (START))), and that, except for I, the value of every variable in (NEXT (NEXT (START))) is equal to its value in (NEXT (START)).

We finally arrive at the CONTINUE statement at label 200 -- the last node in the path. The verification condition for this statement is the conjunction of (1) the instance of the loop assertion at 200 obtained by replacing STATE by the current state term:

```
(AND (EQUAL (PAT (NEXT (NEXT (START)))) (PAT (START)))
     (EQUAL (STR (NEXT (NEXT (START)))) (STR (START)))
     (EQUAL (PATLEN (NEXT (NEXT (START))))
            (PATLEN (START)))
     (EQUAL (STRLEN (NEXT (NEXT (START))))
            (STRLEN (START)))
     (NUMBERP (I (NEXT (NEXT (START)))))
     (NOT (LESSP (I (NEXT (NEXT (START)))) (PATLEN (START))))
     (LESSP (I (NEXT (NEXT (START))))
            (PLUS (PATLEN (START))
                  (SEARCH (PAT (START))
                          (STR (START))
                          (PATLEN (START))
                          (STRLEN (START))
                          1)))
     (IMPLIES (AND (NUM' 'P C)
                   (NOT (EQUAL C 0))
                   (NOT (LESSP (@ASIZE) C)))
              (EQUAL (ELT1 (BLK-DELTA1 (NEXT (NEXT (START)))) C)
                     (DELTA1 (PAT (START))
                             C
                             (PATLEN (START))))))
```

and (2) the lexicographic comparison of the loop clock at 200 (instantiated with the current state term) and the input clock:

```
(LEX (LIST (DIFFERENCE (PLUS 1
                             (STRLEN (START))
                             (PATLEN (START)))
                       (I (NEXT (NEXT (START)))))
           (ADD1 (PATLEN (START))))
     (LIST (PLUS 1
                 (STRLEN (START))
                 (PATLEN (START)))
           0))
```

To prove this we get to use the global assumptions, and the assumptions provided by the previously encountered CALL and assignment statements. In particular, the latter assumptions permit the theorem-prover to reduce terms such as (PAT (NEXT (NEXT (START)))) to (PAT (START)), about which we have assumptions provided by the input assertion.

G. A Comparison With the Toy Version

Our mechanical theorem-prover has proved the verification conditions for SETUP and FSRCH. We will not discuss the other paths through FSRCH. The discussion of the toy version of the algorithm in [5] sketches the proofs that the assertion at the beginning of each path implies the assertion at the end. However, as we have noted, addressing the limitations of a real programming language requires that one consider more than the simple paths from one assertion to the next.

Let us consider the statement labeled 400:

```
400  I = MAX0((I+DELTA1(C)), NEXTI)
```

In our toy version of the problem, the verification condition generator walks through this statement and records the fact that I is the maximum of the present mathematical values of I+DELTA1(C) and NEXTI. The statement requires nothing new for us to prove.

But in a real language, the statement at 400 is a mine field of possible errors. In FORTRAN terms, we have to prove six things to get past this assignment statement:

(1) C is defined.
(2) C is a legal index into DELTA1, i.e., $1 \leq C \leq$ @ASIZE.
(3) DELTA1(C) is defined.
(4) I is defined.
(5) I+DELTA1(C) is expressible.
(6) NEXTI is defined.

We will sketch the proofs of these six facts. First, in the state in which we encounter this statement, C is STR(J). Since J is a legal index into STR and since the input assertion tells us that for all such J, STR(J) is a character code between 1 and @ASIZE, we can conclude that C is defined. We can also conclude that C is a legal index into DELTA1 since @ASIZE is also the size of the DELTA1 array. (Note that the argument that we can use C as an index into DELTA1 is beyond the scope of a standard compiler or type checker.) We can prove that DELTA1(C) is

defined using the invariant about the effect of SETUP on BLK-DELTA1. In particular, DELTA1(C) is equal to the value of the mathematical function DELTA1 applied to certain arguments, and it is easy to show that the function in question always returns a number. We can prove that I and NEXTI are defined from the inner loop invariant. The only remaining problem is to prove that I+DELTA1(C) does not cause an overflow. Here is the proof. The inner invariant tells us that I is positive and less than or equal to STRLEN (ostensibly so that we can use it as an index into STR). As noted above, DELTA1(C) is equal to the value of the mathematical function DELTA1 applied to certain arguments. It is easy to prove by mathematical induction on the size of PAT that the value of the DELTA1 function is nonnegative and less than or equal to PATLEN. Therefore, I+DELTA1(C) is bounded below by 1 and above by STRLEN+PATLEN, which is expressible, by the input assertion.

Thus, we have proved the verification condition required to move past the statement labeled 400. It is encouraging to note that the proof is not deep. Instead, it is merely tedious. But the tedium -- and the responsibility for the logical correctness of the proof -- is the burden of our mechanical theorem-prover, not the user. When the theorem-prover has proved all of the verification conditions generated for SETUP and FSRCH we have gained something real: when executed by any correct FORTRAN processor in an environment that satisfies our input assertion, FSRCH will always terminate, will never cause a run-time error, and computes the correct answer.

XIV Acknowledgments

We would like to thank Paul Yans, of the University of Liège, Belgium, for his useful criticisms of an early draft of this document.

Appendix A

THE BASIC FORTRAN THEORY

Appendix A

THE BASIC FORTRAN THEORY

We construct the basic FORTRAN theory in two stages. First we build the so-called "integer fragment" containing the formal correspondents of the FORTRAN INTEGER and LOGICAL operations, relations, and functions. Then we extend the integer fragment by adding the function symbols required for the REAL, DOUBLE, and COMPLEX types.

We completely specify the integer fragment of the basic FORTRAN theory. We only sketch how to extend the integer fragment to produce the basic FORTRAN theory. We have not yet formalized in our theory the mathematics behind type REAL (and thus also behind types DOUBLE and COMPLEX). However, it is necessary to introduce certain function symbols and assumptions used by the verification condition generator in the handling of REAL, DOUBLE, and COMPLEX expressions.

Definition. integer fragment. The integer fragment is produced by extending the theory in [5] and [6] by adding the following functions, shells, and axioms.

Note. First, we add the function LOGICALP, which recognizes objects of type LOGICAL. The logical operations corresponding to FORTRAN's .AND., .OR., and .NOT. are the functions AND, OR, and NOT, which are already in the primitive theory described in [5] and [6].

```
Definition.
(LOGICALP X)
   =
(OR (EQUAL X (TRUE))
    (EQUAL X (FALSE)))
```

Note. Next we begin the construction of the mathematical functions that correspond to FORTRAN's built-in operations on type INTEGER.

The primitive theory upon which we construct the basic FORTRAN theory already includes the (Peano-like) shell axioms for the natural numbers (recognized by NUMBERP) and the shell axioms for the negatives. The negative number -n is constructed by applying the "constructor" function MINUS to the Peano number n. The negatives are recognized by the function NEGATIVEP. Given a negative representing -n, the function NEGATIVE.GUTS returns the Peano number n. The primitive theory also includes the Peano sum and less than functions, PLUS and LESSP. We begin by defining the remaining elementary functions on natural numbers.

```
Definition.
(DIFFERENCE X Y)
   =
(IF (ZEROP X)
    0
    (IF (ZEROP Y)
        X
        (DIFFERENCE (SUB1 X) (SUB1 Y))))

Definition.
(TIMES X Y)
   =
(IF (ZEROP X)
    0
    (PLUS Y (TIMES (SUB1 X) Y)))

Definition.
(QUOTIENT X Y)
   =
(IF (ZEROP Y)
    0
    (IF (LESSP X Y)
        0
        (ADD1 (QUOTIENT (DIFFERENCE X Y) Y))))

Definition.
(EXPT I J)
   =
(IF (ZEROP J)
    1
    (TIMES I (EXPT I (SUB1 J))))
```

Note. Now, using the negatives and the Peano numbers, we "define" the set of positive and negative integers (often called "Z") by defining

the Boolean function ZNUMBERP to return T or F according to whether its argument is an integer. We then define the standard functions on the integers. These equations define the usual infinite sets of ordered pairs embodying the traditional mathematical notions of integer sum, product, etc. For example, our definition of integer sum specifies the value of the sum of arbitrarily large integers. Because of the finiteness of actual processors, these mathematical notions, by themselves, do not accurately describe the semantics of the corresponding FORTRAN integer operations, and we do not so use them. Instead, we use these mathematical notions in the input/output specifications of those finite FORTRAN operators.

```
Definition.
(ZNUMBERP X)
   =
(OR (NEGATIVEP X) (NUMBERP X))

Definition.
(ZZERO)
   =
(ZERO)

Definition.
(ZPLUS X Y)
   =
(IF
 (NEGATIVEP X)
 (IF
   (NEGATIVEP Y)
   (MINUS (PLUS (NEGATIVE.GUTS X)
                (NEGATIVE.GUTS Y)))
   (IF (LESSP Y (NEGATIVE.GUTS X))
       (MINUS (DIFFERENCE (NEGATIVE.GUTS X) Y))
       (DIFFERENCE Y (NEGATIVE.GUTS X))))
 (IF
   (NEGATIVEP Y)
   (IF (LESSP X (NEGATIVE.GUTS Y))
       (MINUS (DIFFERENCE (NEGATIVE.GUTS Y) X))
       (DIFFERENCE X (NEGATIVE.GUTS Y)))
   (PLUS X Y)))
```

```
Definition.
(ZDIFFERENCE X Y)
   =
(IF
 (NEGATIVEP X)
 (IF (NEGATIVEP Y)
     (IF (LESSP (NEGATIVE.GUTS Y)
                (NEGATIVE.GUTS X))
         (MINUS (DIFFERENCE (NEGATIVE.GUTS X)
                            (NEGATIVE.GUTS Y)))
         (DIFFERENCE (NEGATIVE.GUTS Y)
                     (NEGATIVE.GUTS X)))
     (MINUS (PLUS (NEGATIVE.GUTS X) Y)))
 (IF (NEGATIVEP Y)
     (PLUS X (NEGATIVE.GUTS Y))
     (IF (LESSP X Y)
         (MINUS (DIFFERENCE Y X))
         (DIFFERENCE X Y))))

Definition.
(ZTIMES X Y)
   =
(IF (NEGATIVEP X)
    (IF (NEGATIVEP Y)
        (TIMES (NEGATIVE.GUTS X)
               (NEGATIVE.GUTS Y))
        (MINUS (TIMES (NEGATIVE.GUTS X) Y)))
    (IF (NEGATIVEP Y)
        (MINUS (TIMES X (NEGATIVE.GUTS Y)))
        (TIMES X Y)))

Definition.
(ZQUOTIENT X Y)
   =
(IF (NEGATIVEP X)
    (IF (NEGATIVEP Y)
        (QUOTIENT (NEGATIVE.GUTS X)
                  (NEGATIVE.GUTS Y))
        (MINUS (QUOTIENT (NEGATIVE.GUTS X) Y)))
    (IF (NEGATIVEP Y)
        (MINUS (QUOTIENT X (NEGATIVE.GUTS Y)))
        (QUOTIENT X Y)))

Definition.
(ZEXPTZ I J)
   =
(IF (ZEROP J)
    1
    (ZTIMES I (ZEXPTZ I (SUB1 J)))))
```

Note. (ZEXPTZ I J) returns I raised to the Jth power, provided I is a positive or negative integer and J is a nonnegative integer. In particular, the definition of ZEXPTZ does not handle the case that J is negative -- which involves real (or at least rational) arithmetic. (In fact, since the Peano function ZEROP returns T on all objects other than those constructed by the Peano ADD1 function, (ZEXPTZ I J) is defined to be 1 for negative J.) This is an acceptable definition for ZEXPTZ, which is used in the formalization of INTEGER to INTEGER exponentiation, because the input condition for (I**J), where I and J are of type INTEGER, requires that J be nonnegative.

We next introduce the usual collection of relations on the integers (e.g., "less than," etc.). There is a mild problem caused by our use of the shell principle to introduce the negatives: (MINUS 0) is an object different from 0. Consequently, we cannot use the usual equality predicate as the "meaning" of the FORTRAN relation .EQ. on INTEGERs. Instead, we must define an equality relation on the ZNUMBERPs, called ZEQP, under which (MINUS 0) and 0 are equal. We therefore define (ZNORMALIZE X) to return 0 if X is (MINUS 0) and X if X is any other ZNUMBERP. Our definition of ZEQP ZNORMALIZEs both arguments and then checks equality.

```
Definition.
(ZNORMALIZE X)
   =
(IF (NEGATIVEP X)
    (IF (EQUAL (NEGATIVE.GUTS X) 0) 0 X)
    (FIX X))

Definition.
(ZEQP X Y)
   =
(EQUAL (ZNORMALIZE X) (ZNORMALIZE Y))

Definition.
(ZNEQP X Y)
   =
(NOT (ZEQP X Y))
```

```
Definition.
(ZLESSP X Y)
   =
(IF (NEGATIVEP X)
    (IF (NEGATIVEP Y)
        (LESSP (NEGATIVE.GUTS Y)
               (NEGATIVE.GUTS X))
        (NOT (AND (EQUAL (NEGATIVE.GUTS X) 0)
                  (ZEROP Y))))
    (IF (NEGATIVEP Y) F (LESSP X Y)))

Definition.
(ZLESSEQP X Y)
   =
(NOT (ZLESSP Y X))

Definition.
(ZGREATERP X Y)
   =
(ZLESSP Y X)

Definition.
(ZGREATEREQP X Y)
   =
(NOT (ZLESSP X Y))
```

Note. We now introduce two undefined constants that denote the upper and lower bounds of the machine's integer arithmetic. These bounds are used in the definition of the function EXPRESSIBLE.ZNUMBERP. Our verification condition generator produces formulas that guarantee that the INTEGER constants mentioned in the program text (except in DIMENSION statements) are EXPRESSIBLE.ZNUMBERPs. EXPRESSIBLE.ZNUMBERP is also used in the input condition formulas for the built-in FORTRAN INTEGER operations. For example, the value produced by the FORTRAN expression (X+Y) is a certain sum, as defined by ZPLUS, provided that sum is an EXPRESSIBLE.ZNUMBERP. (Since we guarantee that all constants are expressible and that all built-in operations produce expressible answers, we do not have to consider the possibility that the arguments to a function are inexpressible because there is no way to construct such an integer.)

To reduce the number of trivial verification conditions, such as (EXPRESSIBLE.ZNUMBERP 4), we have added an axiom to the basic FORTRAN theory that is nowhere justified by the definition of FORTRAN, but which is quite reasonable: we assume that the integers between -200 and 200 are all expressible. We know of no FORTRAN processors for which this is false.

Since nothing else is assumed about the range of expressible integers, the reader may wonder how the expressibility conditions are proved (when the value in question is not between -200 and 200). The answer is that they are proved from the input assertions the user supplies on the subprograms being verified. The specifier must state explicitly the size constraints under which his program operates. For example, consider a straightforward "big number" multiplication subroutine for arrays representing digit sequences in an arbitrary base, B. The subroutine must be able to multiply two digits together and obtain their product, to which it must be able to add a "carry" that is less than the base. Thus, the subroutine does not work correctly if one tries to use a base B for which ((B-1)*B) is inexpressible.

```
Undefined Function.
(GREATEST.INEXPRESSIBLE.NEGATIVE.INTEGER)

Undefined Function.
(LEAST.INEXPRESSIBLE.POSITIVE.INTEGER)

Axiom.  INTEGER.SIZE:
(AND
 (ZLESSP (GREATEST.INEXPRESSIBLE.NEGATIVE.INTEGER)
         -200)
 (ZLESSP 200
         (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER)))
```

Note. The above axiom is consistent, since there are no other axioms about these two constants and -201 and 201 are constants in the theory satisfying the two conjuncts.

```
Definition.
(EXPRESSIBLE.ZNUMBERP X)
   =
(AND
 (ZLESSP (GREATEST.INEXPRESSIBLE.NEGATIVE.INTEGER)
         X)
 (ZLESSP X
         (LEAST.INEXPRESSIBLE.POSITIVE.INTEGER)))
```

Note. For each of the finite number of tokens we introduce a constant function of the same name. Each function is required to have an expressible positive integer value. (In particular, 0 must be LESSP the value, from which it can be proved that the value is a positive integer.)

```
For each token, token:

Undefined Function.
(token)

Axiom.  token.POSITIVE
(AND (LESSP 0 (token))
     (EXPRESSIBLE.ZNUMBERP (token)))
```

Note. We now define the mathematical functions computed by the FORTRAN intrinsic functions over type INTEGER.

```
Definition.
(IABS I)
   =
(IF (NEGATIVEP I)
    (NEGATIVE.GUTS I)
    (FIX I))

Definition.
(MOD X Y)
   =
(ZDIFFERENCE X
             (ZTIMES Y (ZQUOTIENT X Y)))

Definition.
(MAX0 I J)
   =
(IF (ZLESSP I J) J I)
```

```
Definition.
(MIN0 I J)
   =
(IF (ZLESSP I J) I J)

Definition.
(ISIGN I J)
   =
(IF (NEGATIVEP J)
    (ZTIMES -1 (IABS I))
    (IABS I))

Definition.
(IDIM I J)
   =
(ZDIFFERENCE I (MIN0 I J))
```

Note. We now define *DEFINEDP*, which is the negation of UNDEFINED, which is the recognizer of a new shell class containing an infinite set of objects. DEFINEDP is used in the enforcement of the FORTRAN requirement that variables be defined before they are used in the primitive arithmetic operations. We introduce it via a shell recognizer rather than as an undefined function so the knowledge that an object satisfies ZNUMBERP (for example) establishes that it is DEFINEDP.

```
Shell Definition.
Add the shell UNDEF of one argument with
recognizer UNDEFINED,
accessor UNDEF.GUTS,
and default value 0.

Definition.
(DEFINEDP X)
   =
(NOT (UNDEFINED X))
```

Note. We introduce four undefined functions: the constant function START, used to denote the arbitrary state at the beginning of the initial paths through a subprogram being verified, and the three functions ELT1, ELT2, and ELT3, used in the denotation of the elements of one-, two-, and three-dimensional arrays.

```
Undefined Function.
(START)

Undefined Function.
(ELT1 A I)

Undefined Function.
(ELT2 A I J)

Undefined Function.
(ELT3 A I J K)
```

Note. We now define the function LEX. Suppose L1 is the list containing the k+1 natural numbers $\langle n_0 \ldots n_k \rangle$ and L2 is the list containing the k+1 natural numbers $\langle m_0 \ldots m_k \rangle$. Then (LEX L1 L2) is (TRUE) if and only if the ordinal $w^k*n_0 + w^{k-1}*n_1 + \ldots + w^0*n_k$ is less than the ordinal $w^k*m_0 + w^{k-1}*m_1 + \ldots + w^0*m_k$. That is, for each natural number k, LEX is the well-founded relation on k+1-tuples of natural numbers induced by the less than relation on w^{k+1}.

```
Definition.
(LEX L1 L2)
  =
(IF (OR (NLISTP L1) (NLISTP L2))
    F
    (OR (LESSP (CAR L1) (CAR L2))
        (AND (EQUAL (CAR L1) (CAR L2))
             (LEX (CDR L1) (CDR L2)))))
```

This completes the specification of the integer fragment of the basic FORTRAN theory.

To construct the basic FORTRAN theory, we extend the integer fragment of the basic FORTRAN theory. The first step is to add notational conventions suitable for admitting noninteger FORTRAN arithmetic constants (e.g., 1.23E45 and (-1.2, 3.4)) as terms. We do not specify those conventions in this document. The second step is the addition of the following functions and axioms concerning the mathematical counterparts of types REAL, DOUBLE, and COMPLEX.

We introduce the monadic Boolean functions RNUMBERP, DNUMBERP, and CNUMBERP to recognize the objects of type REAL, DOUBLE, and COMPLEX, respectively. Thus, (RNUMBERP 0.0E-999) is a theorem.

We introduce the constant functions (RZERO), (DZERO), and (CZERO) to be the zero elements of type REAL, DOUBLE, and COMPLEX, respectively.

We introduce the monadic Boolean functions EXPRESSIBLE.RNUMBERP, EXPRESSIBLE.DNUMBERP, and EXPRESSIBLE.CNUMBERP to play the roles for types REAL, DOUBLE, and COMPLEX (respectively) that EXPRESSIBLE.ZNUMBERP plays for type INTEGER. For example, if a program mentions the REAL 0.2E-999, then one of the verification conditions will include the proposition:

(EXPRESSIBLE.RNUMBERP 0.2E-999).

Thus, whoever specifies the operations on the REALs need not include (in the input condition formulas for those operations) the requirement that each REAL input is expressible provided he guarantees that no such operation can generate an inexpressible REAL result.

We introduce the dyadic function symbols RPLUS, RTIMES, RDIFFERENCE, RQUOTIENT, RLESSP, RLESSEQP, REQP, RNEQP, RGREATEREQP, and RGREATERP.

We introduce the dyadic function symbols DPLUS, DTIMES, DDIFFERENCE, DQUOTIENT, DLESSP, DLESSEQP, DEQP, DNEQP, DGREATEREQP, and DGREATERP.

We introduce the dyadic function symbols CPLUS, CTIMES, CDIFFERENCE, CQUOTIENT, CEQP, and CNEQP.

We introduce the dyadic function symbols REXPTZ, DEXPTZ, CEXPTZ, REXPTR, REXPTD, DEXPTR and DEXPTD. (ZEXPTZ, the INTEGER to INTEGER exponentiation function was handled in the integer fragment.)

For each intrinsic function pattern with name fn and n arguments (except those intrinsic functions already introduced in the integer fragment, IABS, MOD, MAX0, MIN0, ISIGN, and IDIM), we introduce the n-ary function symbol fn, eventually to be defined to represent the mathematical function specified in the FORTRAN definition for the intrinsic function named fn.

Note. We do not specify in this document what the properties of the REAL, DOUBLE, or COMPLEX operations are. Intuitively, if the REAL variables X and Y have the values x and y at run time, then the value of the FORTRAN expression (X+Y) is understood to be (RPLUS x y), provided x and y satisfy the input conditions specified for RPLUS.

Appendix B

INPUT CONDITION FORMULAS

Appendix B

INPUT CONDITION FORMULAS

The table below gives the "input condition formula" for each built-in INTEGER operation, relation, and intrinsic function. The formula is used in the definition of the "input condition" for an expression. Eventually such formulas must be supplied for all the REAL, DOUBLE, and COMPLEX routines as well. At the moment, the formula for these routines is (FALSE), which means that a program involving REAL, DOUBLE PRECISION, or COMPLEX arithmetic cannot be proved correct by our system since the input conditions could never be established.

Rather than give the input condition formula for the FORTRAN symbol that appears in an expression e, e.g., + or .LE., we give the formula for the function symbol of the term [e], e.g., ZPLUS or ZLESSEQP, since the types of the arguments determine the precise interpretation of the symbol.

Definition. input condition formula. The input condition formulas for certain function symbols in the basic FORTRAN theory are specified by the table below. The input condition formula for a name not in the left-hand column is (FALSE).

name	input condition formula
ZPLUS	(EXPRESSIBLE.ZNUMBERP (ZPLUS (I STATE) (J STATE)))
ZDIFFERENCE	(EXPRESSIBLE.ZNUMBERP (ZDIFFERENCE (I STATE) (J STATE)))
ZTIMES	(EXPRESSIBLE.ZNUMBERP (ZTIMES (I STATE) (J STATE)))
ZQUOTIENT	(AND (ZNEQP (J STATE) 0)) (EXPRESSIBLE.ZNUMBERP (ZQUOTIENT (I STATE) (J STATE))))

```
ZEXPTZ          (AND (NOT (AND (ZEQP (I STATE) 0)
                               (ZEQP (J STATE) 0)))
                     (ZLESSP -1 (J STATE))
                     (EXPRESSIBLE.ZNUMBERP
                      (ZEXPTZ (I STATE) (J STATE))))

ZLESSP          (TRUE)

ZLESSPEQP       (TRUE)

ZEQP            (TRUE)

ZNEQP           (TRUE)

ZGREATEREQP     (TRUE)

ZGREATERP       (TRUE)

NOT             (TRUE)

AND             (TRUE)

OR              (TRUE)

IABS            (EXPRESSIBLE.ZNUMBERP
                 (IABS (I STATE)))

MOD             (AND (ZNEQP (J STATE) 0)
                     (EXPRESSIBLE.ZNUMBERP
                      (MOD (I STATE) (J STATE))))

MAX0            (TRUE)

MIN0            (TRUE)

ISIGN           (AND (ZNEQP (J STATE) 0)
                     (EXPRESSIBLE.ZNUMBERP
                      (ISIGN (I STATE) (J STATE))))

IDIM            (EXPRESSIBLE.ZNUMBERP
                 (IDIM (I STATE) (J STATE)))
```

REFERENCES

1. American National Standards Institute, Inc., American National Standard Programming Language FORTRAN, ANSI X3.9-1978, 1430 Broadway, New York, New York 10018, April 3, 1978.

2. R. B. Anderson, Proving Programs Correct, (John Wiley & Sons, New York, New York, 1979).

3. J. Backus, "The History of FORTRAN I, II, and III," SIGPLAN Notices, 13(8) pp. 165-180 (August, 1978). Copyright 1978, Association for Computing Machinery, Inc., reprinted by permission.

4. R. S. Boyer and J S. Moore, "A Fast String Searching Algorithm," Commun. Assoc. Comput. Mach., 20(10), pp. 762-772 (1977).

5. R. S. Boyer and J S. Moore, A Computational Logic, (Academic Press, New York, New York, 1979).

6. R. S. Boyer and J S. Moore, "Metafunctions: Proving them Correct and Using them Efficiently as New Proof Procedures," NOTE TO EDITOR: Please insert here a reference to the other article in this book by Boyer and Moore.

7. W. S. Brown, "A Realistic Model of Floating-Point Computation," in Mathematical Software III, J. R. Rice, ed., pp 343-360 (Academic Press, New York, New York, 1977).

8. R. W. Floyd, "Assigning Meanings to Programs," Mathematical Aspects of Computer Science, Proc. Symp. Appl. Math. Vol. XIX, pp. 19-32 (American Mathematical Society, Providence, Rhode Island, 1967).

9. H. H. Goldstine and J. von Neumann, "Planning and Coding Problems for an Electronic Computing Instrument," Part II, Vol. 1, 1947, reproduced in John von Neumann Collected Works, Vol. V, p. 113 (Pergamon Press, Oxford, 1961).

10. J. C. King, "A Program Verifier," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (September, 1969).

11. Z. Manna, Mathematical Theory of Computation, (McGraw-Hill, Inc, New York, New York, 1974).

12. United States of America Standards Institute, USA Standard FORTRAN, USAS X3.9-1966, 10 East 40th Street, New York, New York 10016, March 7, 1966.

INDEX

DISTRIBUTION LIST

Office of Naval Research — 2 copies
Information Systems Program
Code 437
Arlington, VA. 22217

Office of Naval Research — 1 copy
Branch Office, Boston
495 Summer Street
Boston, MASS. 02210

Office of Naval Research — 1 copy
Branch Office, Chicago
536 South Clark Street
Chicago, ILL. 60605

Office of Naval Research — 1 copy
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA. 91106

New York Area Office — 1 copy
715 Broadway - 5th Floor
New York, N.Y. 10003

Assistant Chief for Technology — 1 copy
Office of Naval Research
Code 200
Arlington, VA. 22217

Naval Research Laboratory — 6 copies
Technical Information Div.,
Code 2627
Washington, D.C. 20375

Dr. A. L. Slafkosky — 1 copy
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380

Office of Naval Research — 1 copy
Code 455
Arlington, VA. 22217

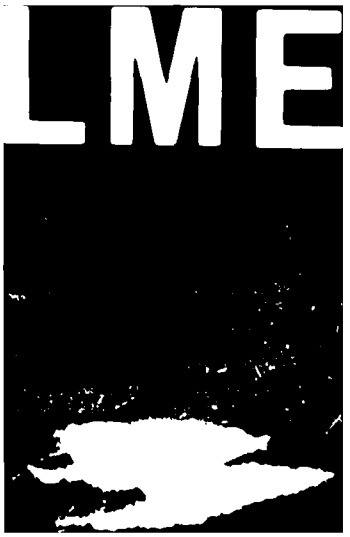
Office of Naval Research — 1 copy
Code 458
Arlington, VA. 22217

Naval Elec. Laboratory Center — 1 copy
Advanced Software Tech. Div.
Code 5200
San Diego, CA. 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Res. & Dev. Center
Computation & Math. Dept.
Bethesda, MD. 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, D.C. 20350

Officer-in-Charge — 1 copy
Naval Surface Weapons Center
Dahlgren Laboratory
Dahlgren, VA. 22448
Attn: Code KP


LME