A Definition of A I D

THE ATTEST INTERFACE DESCRIPTION LANGUAGE

Daryl Winters
Neal Ogden
Lori Clarke

COINS Technical Report 78-15
DRAFT - December 1, 1978.

Computer and Information Science Department
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

This work was supported by the National Science Foundation under grant # NSFMCS 77-02101 and the U.S. Air Force under grant # AFOSR 77-3287.

Abstract

The Automatic Test Enhancement System (ATTEST) is a symbolic execution system which assists the user in developing and testing software written in ANSI FORTRAN. The ATTEST Interface Description Language (AID) is a hierarchically structured command language which enables the user to control the operation of ATTEST. This report describes the use of AID as an interface between ATTEST and the user of the system. The syntax of AID is defined along with semantic descriptions and examples of each type of AID command.

Table of Contents

1. AID Definition

The Automatic Test Enhancement System (ATTEST) is a symbolic execution system which assists the user in developing and testing software written in ANSI FORTRAN. The ATTEST Interface Description Language (AID) is a hierarchically structured command language which enables the user to control the operation of ATTEST. This report describes the use of AID as an interface between ATTEST and the user of the system. The syntax of AID is defined along with semantic descriptions and examples of each type of AID command.

1.1. General Concepts

A FORTRAN program is generally composed of a main routine and a number of subsidiary routines (either SUBROUTINE or FUNCTION routines). Each routine is composed of a number of statements which describe the sequence of computations to be performed. Some statements transfer control between statements - either conditionally (IF, computed GOTO, etc.) or unconditionally (GOTO). ATTEST represents each routine as a directed graph, known as a control flow graph, in which nodes represent statements and edges represent possible transfer of control between two statements. By following the programs control flow graph from some starting point to a STOP statement, a path through the program is generated. This path represents a possible execution of the program. ATTEST, very generally, evaluates a number of such paths through the program in an attempt to generate a set of test data which would execute those paths.

AID attempts to reflect the natural execution sequence in its hierarchical structure. At the outermost level, the description of an ATTEST session is composed of a description of a number of paths through a program which will be processed by ATTEST. Each path is in turn described by the routines which would be encountered along that path. Since a particular path may encounter a single routine several times, a routine can be described by the specific entries into that routine. At each level, AID procedural commands may be used to control the execution through that particular level. Each level describes a "scope" which is simply the range over which the sequence of commands will be in effect - either a path, a routine on that path, or a specific entry to that routine may be a scope. Certain commands, known as addressable commands, must be directed to a specific point in the control flow graph. That point is known as an address. Addressable commands must be addressed to a specific point because they are dependant upon the particular path which was followed to reach that address.

1.2. Symbolic Execution

ATTEST is a symbolic execution system. AID enables the user to control the processing of that symbolic execution. In symbolic execution (as in normal execution), the set of program variables is divided into three classes: input variables which directly recieve values from input units via READ statements, intermediate variables which (perhaps indirectly) recieve values from input variables via assignment statements, and output variables which display the programs results through output units

via WRITE statements. In normal execution, input variables recieve actual numeric or hollerith values. Each subsequent arithmetic operation and assignment statement produces further numeric or hollerith values. In this way, all output values display results in terms of numeric or hollerith values.

In symbolic execution, on the other hand, input variables recieve a "symbolic" value representing the whole class of actual values which may be recieved during normal execution. Intermediate variables would then recieve symbolic expressions representing the arithmetic operations performed on the symbolic values of variables contained in the expression on the right hand side of the assignment statement. Thus, output variables would display symbolic expressions in terms of the symbolic values used to compute the value of that output variable.

1.2.1. Symbolic Input/Output Units

In order to allow symbolic execution to proceed in its most general sense (with all input/output operations performed symbolically), all FORTRAN input/output units may be described symbolically. Normally, variables recieve actual numeric or hollerith values (depending on the FORMAT control). By describing the data in input/output units symbolically, a program can be symbolically executed and recieve values which "represent" the actual values recieved during normal execution.

Since the data is represented symbolically, no FORMAT control will be used when reading from or writing to a symbolic unit. Instead, each

separate input/output operation will read a single "record" from the unit. Each record consists of strings of symbolic names (perhaps combined with "symbolic" arithmetic operators) each separated by comma "," symbols. An example of a READ statement and a single record of a symbolic input/output unit are shown below.

```
READ (1,100) I,J,K
100 FORMAT (I3,2X,I10,/,I10)

INDEX, MAXIMUM, NEXT+INDEX
```

After symbolically executing the above READ statement, the variables would contain the values shown below (note that FORMAT control is effectively ignored).

```
I = "INDEX"
J = "MAXIMUM"
K = "NEXT+INDEX"
```

1.3. Command Entry

An AID command may be entered in one of two ways - as an external command or an embedded command. In the remaining sections, all descriptions of the format of AID commands are the same as for external commands.

1.3.1. External Commands

An external command is entered either through an external command file or through an interactive terminal. In either case, each command line may contain up to 150 characters. Each computer system running ATTEST will have a separate method for associating the AID command files with the ATTEST system. Each will be described in an addendum to this report.

1.3.2. Embedded Commands

An embedded command is contained in the source text that is submitted to ATTEST. The embedded commands are disguised as FORTRAN comment lines by using "C AID" in columns 1 to 5. The remainder of the line (columns 7 to 80) are assumed to contain AID commands.

The following is an example of an embedded AID command.

```
C AID COMMAND SEQUENCE
```

In the above example, "C AID" would be recognized as the special AID flag and ignored while "COMMAND SEQUENCE" would be checked for a legal AID command. (Note that any FORTRAN comment line beginning with "C AID" will be assumed to contain an AID command).

1.4. Lexical Elements

A lexical element is the fundamental unit of AID - all declarations and commands are generated from a series of lexical elements. Each element is an indivisible unit and must appear on a single line. Spaces may be

inserted freely between lexical elements. At least one space is required between all lexical elements on a line unless a required special symbol is used.

A lexical element is formed using only the following FORTRAN characters.

[1] Alphabetic Characters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

[2] Numeric Characters

0 1 2 3 4 5 6 7 8 9

[3] Special Characters

= + - * / ( ) , . $

[4] Blank Character

(space)

1.4.1. Special Symbols

A special symbol is a lexical element formed with one or two special characters. A special symbol must fit entirely on a single line; no interspersed blanks are allowed. The following is a list of all special symbols recognized by AID:

= + - * / ( ) , . $ .. --

1.4.2. Identifiers

An identifier is a lexical element formed with a sequence of one to six alphabetic and numeric characters, the first of which must be alphabetic. An identifier must fit entirely on a single line; no interspersed blanks or special characters are allowed.

1.4.3. Keywords

A keyword is a lexical element formed with a sequence of one to ten alphabetic characters. A keyword is not a reserved word and may be utilized freely as an identifier. Any keyword may be abbreviated to its first four letters (if the keyword is longer than four letters); only the first four letters of a keyword are considered significant (the remaining letters are ignored). Although some keywords are shown explicitly as either plural or singular, since only the first four letters are considered significant, any keyword (longer than four letters) can be made plural to make a command more grammatically correct.

The following is a list of all keywords recognized by AID:

ABORT	ELSE	POSITION
AFTER	END	PREDECESSOR
ALL	ENTRY	PROCESS
ARGUMENT	EXIT	RANGE
ARITHMETIC	EXPLAIN	REFERENCE
ASSERT	EXTERNAL	RESUME
ASSIGN	EXTERNALS	RETURN
ASSIGNMENT	FINAL	ROUTINE
ASSUME	FIND	SOLUTION
ATTEMPT	GLOBAL	SOME
BEFORE	GO	START
BLOCK	HEADER	STOP
BLOCKS	HELP	STRING
BRANCHES	IN	SUBSCRIPT

BREAK	INDEX	SUCCESSOR
CONDITION	INPUT	TIMES
CONDITIONAL	INITIAL	TRACE
CONNECT	LABEL	TO
CONSTRAINTS	NO	UNIT
CONTINUE	OR	VALUE
DEFINITION	OTHERWISE	WEIGHT
DESCRIPTION	OUTPUT	WHEN
DISPLAY	PARAMETER	WITH
DO	PATH	

The following keywords are used in the HELP facility of AID. These keywords may be further abbreviated to their first letter.

EXPLAIN
FIND
PREDECESSOR
SUCCESSOR

1.4.4. Symbolic Names

A symbolic name is a lexical element formed with a sequence of one to thirty alphabetic and numeric characters, the first of which must be alphabetic. A symbolic name is distinct from any identifier containing the same characters.

1.4.5. Comments

A comment may be placed in an AID command sequence to improve readability. A comment is denoted by the special symbol "--". When the special symbol is recognized, the remainder of the line is ignored.

The following is an example of an AID command containing a user comment.

COMMAND SEQUENCE -- USER COMMENT

In the above example, the sequence "COMMAND SEQUENCE" would be checked for a legal AID command while "-- USER COMMENT" would be ignored.

1.5. Syntax Notation

In the remaining sections, a simple variant of Backus-Naur Form (BNF) notation is used to define the syntax of AID. A meta-symbol is a symbol used in the BNF to describe the syntax of AID and is not part of the AID language. The following characters are meta-symbols:

< > [ ] { } | :=

The following conventions are used in the BNF notation in the remaining sections.

[1] Upper case words (possibly containing underscore characters "_") denote AID recognized keywords. The underscore character is used to indicate that zero or more spaces may be used in that position. If zero spaces are used, the resulting combined keyword may also be abbreviated to the first four characters.

[2] Lower case words (possibly containing underscore characters "_") surrounded by the symbols "<" and ">" denote syntactic categories.

[3] Lower case words (possibly containing underscore characters "_") not surrounded by the symbols "<" and ">" denote FORTRAN

syntactic categories which will be informally defined.

[4] Syntactic units surrounded by square brackets "[" and "]" denote optional items.

[5] Syntactic units surrounded by braces "{" and "}" denote items which may be repeated zero or more times.

[6] Syntactic units separated by a bar character "|" denote alternate items. Any one (only one) of the alternate items may be used to expand the syntactic category on the left of the definition symbol.

[7] The definition symbol ":=" is used to indicate that the syntactic unit to the right of the definition symbol may be used to expand the syntactic category on the left of the symbol.

2. Scope Declarations

A scope declaration is used to describe a portion of an ATTEST session. A scope is used to limit the range of a particular group of commands to some logical or physical portion of a FORTRAN program. Commands may be limited to particular paths (generated by ATTEST) with a path declaration. Within each path, commands may be limited to a particular routine with a routine declaration. Commands may be further limited to a specified entry to a given routine with an entry declaration. Finally, commands may be limited to a single point in a routine by an address declaration.

Embedded commands may be explicitly "scoped" by specifying the scope with a complete scope declaration (including both "BEGIN" and "END"). Embeded commands may also be implicitly "scoped" by simply entering the command in the position desired. The scope of this type of embedded command is then the routine it is in and (for addressable commands) the address is before the statement following the command.

The syntax of scope declaration is:

<scope_declaration> := BEGIN [AID] {<path_unit>} END [AID]

<path_unit> := [<path_declaration>] {<routine_unit>}

<routine_unit> := [<routine_declaration>] {<entry_unit>}

<entry_unit> := [<entry_declaration>] <command_unit>

<command_unit> := [<procedural_group>] {<address_unit>}

<procedural_group> := <procedural_command> {<procedural_command>}

<address_unit> := <address_declaration> {<addressable_command>}

Each type of declaration and command will be explained separately. Before describing each command and declaration, however, an important keyword must be explained first - the "NO" keyword. The "NO" keyword may preface any scope declaration or command. By prefacing a scope declaration with the "NO" keyword, all commands in the entire scope will be ignored. By prefacing a command with the "NO" keyword the effect of a previous command can be nullified (the net result is the same as if the original command was never entered). In this way the command file can be "edited" to remove certain commands without permanently removing that command from the command file. Thus, the command can be easily "re-instated" at a later date by simply removing the "NO". Similiarly, an entire scope declaration can be temporarily "turned off".

2.1. Path Declarations

A path declaration is used to define a subset of paths and the particular actions which will apply to those paths. The declaration is denoted by the keyword "PATH" followed by a path list. A path is denoted by an integer (ie. 1, 2, ..., n) which indicates the order that the path will be processed by ATTEST relative to other paths. (Note that paths may be "defined" in any order but will be processed in the order denoted by their ordinal value.) If the path declaration is omitted, the default path declaration "PATH *" is assumed. All succeeding commands (until the next path declaration) will be invoked only on the specified paths. If the optional "NO" keyword is used, the specified paths (including the succeeding commands) will be ignored.

The syntax of a path declaration is:

<path_declaration> := [NO] PATH <path_list>

<path_list> := <path_scope> {, <path_scope>}

<path_scope> := * | <path_name>

<path_name> := integer [.. <path_range>]

<path_range> := * | integer

In the following examples of path declarations, "i" and "j" are used to represent any positive integer.

[1] PATH *

The declaration defines a "global" path scope. The succeeding commands will apply to all paths defined.

[2] PATH i

The declaration defines a specific path scope numbered "i". The succeeding commands will apply only to path "i".

[3] PATH i , j

The declaration defines two individual path scopes numbered "i" and "j". The succeeding commands will apply to both paths "i" and "j".

[4] PATH i .. j

The declaration defines a sequence of path scopes numbered "i", "i+1", ..., "j". The succeeding commands will apply to path "i", path "i+1", ..., path "j".

[5] PATH i .. *

The declaration defines a sequence of path scopes numbered "i", "i+1", ..., "n". The succeeding commands will apply to path "i", path "i+1", ..., path "n" (where "n" is the maximum path number allowed by ATTEST).

2.2. Routine Declaration

A routine declaration is used to define the particular actions which will apply to a specific routine on a path. The declaration is denoted by the keyword "ROUTINE" followed by a routine list. If the routine declaration is omitted, the default routine declaration "ROUTINE *" is assumed. All succeeding commands (until the next routine or path declaration) will be invoked only in the specified routines. If the optional "NO" keyword is used, the specified routines (including the succeeding commands) will be ignored.

The syntax of a routine declaration is:

<routine_declaration> := [NO] ROUTINE <routine_list>
<routine_list> := <routine_scope> {, <routine_scope>}
<routine_scope> := * | <routine_name>
<routine_name> := identifier [.. <routine_limit>]
<routine_limit> := identifier | <routine_range>
<routine_range> := * | integer

In the following examples of routine declarations, "x" and "y" are used to represent any routine name, and "i" is used to represent any positive integer.

[1] ROUTINE *

The declaration defines a "global" routine scope. The succeeding commands will apply to all routines.

[2] ROUTINE x

The declaration defines a specific routine scope referenced by the name "x". The succeeding commands will apply only to routine "x".

[3] ROUTINE x , y

The declaration defines two individual routine scopes referenced by the names "x" and "y". The succeeding commands will apply to both routines "x" and "y".

[4] ROUTINE x .. y

The declaration defines a dynamic routine scope bounded by the routines "x" and "y". The succeeding commands will apply to all routines entered after routine "x" and until "y" is entered (including both "x" and "y").

[5] ROUTINE x .. *

The declaration defines a dynamic routine scope. The succeeding commands will apply to all routines entered after routine "x" is entered and until "x" returns to its calling routine.

[6] ROUTINE x .. i

The declaration defines a dynamic routine scope bounded by the routine "x" and the "i"-th successive routine called from "x". The succeeding commands will apply to all routines entered after

"x" and until a routine at the "i"-th call from "x" is entered (including both "x" and the "i"-th called routine) or until the end of the path.

2.3. Entry Declaration

An entry declaration is used to restrict a set of actions to a specific entry to a routine on a path. The declaration is denoted by the keyword "ENTRY" followed by an entry list. An entry is denoted by an integer (ie. 1, 2, ..., n) which indicates the entry to a given routine on a particular path. If the entry declaration is omitted, the default entry declaration "ENTRY *" is assumed. All succeeding commands (until the next entry, routine, or path declaration) will be invoked only in the specified entries to the routine. If the optional "NO" keyword is used, the specified entries (including the succeeding commands) will be ignored.

The syntax of an entry declaration is:

<entry_declaration> := [NO] ENTRY <entry_list>

<entry_list> := <entry_scope> {, <entry_scope>}

<entry_scope> := * | <entry_name>

<entry_name> := integer [.. <entry_range>]

<entry_range> := * | integer

In the following examples of entry declarations, "i" and "j" are used to represent any positive integer.

[1] ENTRY *

The declaration defines a "global" entry scope. The succeeding commands will apply to all entries into the routine.

[2] ENTRY i

The declaration defines a specific entry scope numbered "i". The succeeding commands will apply only to entry "i" into the routine.

[3] ENTRY i , j

The declaration defines two individual entry scopes numbered "i" and "j". The succeeding commands will apply to both entries "i" and "j" to the routine.

[4] ENTRY i .. j

The declaration defines a sequence of entry scopes numbered "i", "i+1", ..., "j". The succeeding commands will apply to entry "i", entry "i+1", ..., entry "j" into the routine.

[5] ENTRY i .. *

The declaration defines a "global" entry scope. The succeeding commands will apply to entry "i", entry "i+1", ..., entry "n" (where "n" is the maximum entry).

3. Procedural Commands

A procedural command is associated with a particular scope by the preceeding sequence of path, routine, and entry declarations. A procedural command cannot be addressed to a particular address - it modifies or controls global events within the specified scope. A procedural command is denoted by one of the keywords "TRACE", "WEIGHT", "CONNECT", "ATTEMPT", or "START" followed by information specific to that command. The command will be in effect whenever the specified scope is entered and will be ignored whenever the scope is exited.

The syntax of a procedural command is:

```
<procedural_command> := <trace_command> |
                        <weight_command> |
                        <connect_command> |
                        <attempt_command> |
                        <start_command>
```

Each type of procedural command will be explained separately.

3.1. Trace Commands

A trace command is used to display information about actions which may occur at an unknown time or place in the specified scope. The information will be presented as soon as it is gathered. If the optional "NO" keyword is used, the command will be ignored.

The syntax of a trace command is:

<trace_command> := [NO] TRACE <trace_list>

<trace_list> := <trace_category> {, <trace_category>}

<trace_category> := <control_flow> |
<constraint_flow> |
<data_flow>

Each type of trace category will be explained separately.

In the following examples of trace commands, "a" and "b" are used to represent any trace category.

[1] TRACE a

The command indicates that information of the category "a" will be displayed.

[2] TRACE a , b

The command indicates that information of both category "a" and "b" will be displayed.

3.1.1. Control Flow

A control flow trace command is used to display transfers of control within the current scope. Any transfer of the specified type will be displayed when the transfer occurs.

The syntax of a control flow command is:

<control_flow> := EXTERNALS | BRANCHES | BLOCKS

The following are examples of control flow commands.

[1] TRACE EXTERNALS

The trace command indicates that all transfers of control within the current scope to external routines will be displayed. (Note that external routines includes FORTRAN intrinsic and basic external functions as well as user-defined external routines.)

[2] TRACE BRANCHES

The trace command indicates that all transfers of control within the current scope of a non-sequential nature will be displayed. (Note that this implies all EXTERNALS will be displayed.)

[3] TRACE BLOCKS

The trace command indicates that all transfers of control within the current scope (including sequential) will be displayed. (Note that this implies all EXTERNALS and BRANCHES will be displayed.)

3.1.2. Constraint Flow

A constraint flow trace command is used to display a record of all constraints generated within the current scope. Whenever the flow of control is dependant upon the value of an input variable, the constraints imposed on that variable will be displayed.

The syntax of a constraint flow trace command is:

<constraint_flow> := CONSTRAINTS

The following is an example of a constraint flow trace command.

[1] TRACE CONSTRAINTS

The trace command indicates that all constraints generated in the current scope will be displayed. A constraint is generated whenever the flow of control is dependant on the value of an input variable.

3.1.3. Data Flow

A data flow trace command is used to display data flow within or into and out of the current scope. Data flow may occur through parameter or global variables (external data flow) through input/output units (unit data flow) or within the specified scope (internal data flow). The data flow category is denoted by one of the keywords "PARAMETER", "GLOBAL", "LOCAL", or "UNIT" followed by information specific to that category.

The syntax of an data flow trace command is:

```
<data_flow> := <external_data_flow> |
               <internal_data_flow> |
               <unit_data_flow>
```

Each type of data flow trace command will be explained separately.

3.1.3.1. External Data Flow

An external data flow trace command is used to display data flow into or

out of the current scope through parameter or global variables. An external type keyword is used to specify whether data flow through parameter variables ("PARAMETER") or global variables ("GLOBAL") should be displayed. After entry to the scope, all data flow of the specified type into the scope will be displayed. Before exit from the scope, all data flow of the specified type out of the scope will be displayed. In addition, any call to an external routine will cause data flow of the specified type into or out of the scope to be displayed.

The syntax of a external data flow trace command is:

<external_data_flow> := <external_type> <external_phrase>

<external_type> := PARAMETER | GLOBAL

<external_phrase> := <variable_name> | <external_group>

<external_group> := (<variable_name> {, <variable_name>})

<variable_name> := * | identifier

In the following examples of external data flow trace commands, "t" is used to represent an external type keyword ("PARAMETER" or "GLOBAL"), and "v" and "w" are used to represent any variable name.

[1] TRACE t *

The trace command indicates that external data flow of the command "t" into or out of the current scope for all variables will be displayed.

[2] TRACE t v

The trace command indicates that external data flow of the category "t" into or out of the current scope for the variable

ent scope ofll be displayed.

[3] TRACE t (v , w)

The trace category indicates that a record of external data flow of the category "t" into or out of the current scope both variables "v" and "w" will be displayed.

3.1.3.2. Local Data Flow

A local data flow command is used to display a record of data flow within the current scope. Each operation requiring the value of an external or supplying a value to the variable will be displayed.

The syntax of a local data flow command is:

<local_data_flow> := LOCAL <local_phrase>

<local_phrase> := <local_name> | <local_group>

<local_group> := (<local_name> {, <local_name>})

<local_name> := * | variable

In the following examples of local data flow trace commands, "v" and "w" are used to represent any variable name.

[1] TRACE LOCAL *

The trace command indicates that local data flow within the current scope will be traced. Any operation requiring or supplying a value to any variable will be displayed.

[2] TRACE LOCAL v

The trace command indicates that local data flow within the current scope will be traced. Any operation requiring or supplying a value to the variable "v" will be displayed.

[3] TRACE LOCAL i

The trace command indicates that local data flow within the current scope will be traced. Any operation requiring or supplying a value to the variable "i" will be displayed.

[4] TRACE LOCAL (v , w)

The trace command indicates that local data flow within the current scope will be traced. Any operation requiring or supplying a value to the variable "v" or "w" will be displayed.

3.1.3.3. Unit Data Flow Command

A unit data flow trace command is used to display a record of data flow into or out of the current scope through FORTRAN input/output units. Each operation on, and all values written to or read from the specified units will be displayed.

The syntax of a unit data flow trace command is:

<unit_data_flow> := UNIT <unit_phrase>

<unit_phrase> := <unit_name> | <unit_group>

<unit_group> := (<unit_name> {, <unit_name>})

<unit_name> := * | <unit_value>

<unit_value> := identifier | integer

In the following examples of unit data flow trace commands, "v" and "w" are used to represent any variable name which contains a FORTRAN unit number, and "i" is used to represent any FORTRAN unit number.

[1] TRACE UNIT *

The trace command indicates that external data flow into or out of the current scope through all input/output units will be displayed.

[2] TRACE UNIT v

The trace command indicates that external data flow into or out of the current scope through the input/output unit "v" will be displayed.

[3] TRACE UNIT i

The trace command indicates that external data flow into or out of the current scope through the input/output unit "i" will be displayed.

[4] TRACE UNIT (v , w)

The trace command indicates that external data flow into or out of the current scope through both input/output units "v" and "w" will be displayed.

3.2. Weight Command

A weight command is used to assign a specific weight to elements of the control flow graph of the current scope. Weights are positive or

negative integer values. Initially, each graph element has a weight of "1" (indicating that all elements are equally likely to be chosen). Modifying the weight of an element to a greater positive value increases the likelihood that an element will be exercised more often than other elements with less weight. Modifying the weight of an element to a negative value inhibits ATTEST from exercising that element. If the optional "NO" keyword is used, the command will be ignored.

The syntax of a weight command is:

<weight_command> := [NO] WEIGHT <weight_list> WITH <weight_value>
<weight_list> := <graph_element> {, <graph_element>}
<graph_element> := <graph_edge> | <graph_group> | <graph_node>
<graph_group> := (<graph_edge>)
<graph_edge> := <graph_node> - <graph_node>
<graph_node> := <statement_designator>
<weight_value> := [-] integer

In the following examples of weight commands, "s" and "t" are used to represent any statement designator, and "i" is used to represent any positive integer.

[1] WEIGHT s WITH i

The command indicates that the graph element identified by "s" will be given the weight "i".

[2] WEIGHT s WITH -i

The command indicates that the graph element identified by "s" will be given the weight "-i". (Note that this will inhibit

ATTEST from exercising the specified graph element.)

[3] WEIGHT s , t WITH i

The command indicates that the two graph elements identified by "s" and "t" will both be given the weight "i"

3.3. Connect Command

A connect command is used to specify the particular graph elements which will constitute a path through the current scope. The list of graph elements does not need to completely specify the exact path to follow - only the portions of the path which should be followed. Note that the order in which the graph elements are specified is the same order in which they will appear on a completed path (perhaps with other elements intersperced). A repetition grouping may be used to indicate that the specified group should appear a number of times (this could be used to control the execution of a loop). If the optional "NO" keyword is used, the command will be ignored.

The syntax of a connect command is:

<connect_command> := [NO] CONNECT <connect_list> [OR ABORT]

<connect_list> := <path_component> {, <path_component>}

<path_connect> := <graph_element> | <repeat_group>

<graph_element> := <graph_edge> | <graph_group> | <graph_node>

<graph_group> := (<graph_edge>)

<graph_edge> := <graph_node> - <graph_node>

<graph_node> := <statement_designator>

<repeat_group> := <repeat_value> (<component_list>)

<repeat_value> := * | <repeat_scope>

<repeat_scope> := integer [.. <repeat_limit>]

<repeat_limit> := * | integer

In the following examples of connection commands, "s" and "t" are used to represent any statement designator, and "i" and "j" are used to represent any positive integer.

[1] CONNECT s

The command indicates that the current path should include the graph element identified by "s".

[2] CONNECT s , t

The command indicates that the current path should include the two graph elements identified by "s" and "t".

[3] CONNECT * (s)

The command indicates that the graph element identified by "s" should appear on the current path at least one time.

[4] CONNECT i (s)

The command indicates that the graph element identified by "s" should appear on the current path "i" times.

[5] CONNECT i .. j (s)

The command indicates that the graph element identified by "s" should appear on the current path at least "i" and no more than "j" times.

[6] CONNECT i .. * (s)

The command indicates that the graph element identified by "s" should appear on the current path "i" or more times.

3.4. Attempt Command

An attempt command is used to control the number of attempts to generate a complete path through the current scope. When a possible path cannot be successfully completed, a new attempt will be made to generate a complete path until the specified number of attempts is exhausted (or until no more paths can be tried). If the attempt command is omitted, the default command "ATTEMPT * TIMES" is assumed. If the optional "NO" keyword is used, the specified addresses (including the succeeding addressable commands) will be ignored.

The syntax of an attempt command is:

<attempt_command> := [NO] ATTEMPT <attempt_limit> TIMES

<attempt_limit> := * | integer

In the following examples of attempt commands, "i" is used to represent any integer.

[1] ATTEMPT * TIMES

The command indicates that an attempt will be made to generate a complete path through the current scope until all possible paths have been tried.

[2] ATTEMPT i TIMES

The command indicates that an attempt will be made to generate a complete path through the current scope until "i" possible paths have been tried.

3.5. Start Command

A start command is used to indicate that the path is to begin at a specific routine. If the start command is omitted, the default starting point "START *" is assumed. When the path is processed, control will be transfered to the specified routine. If the optional "NO" keyword is used, the command will be ignored.

The syntax of a start command is:

<start_command> := [NO] START <start_location>

<start_location> := * | identifier

In the following examples of start commands, "x" is used to represent any routine name.

[1] START *

The command indicates that execution of the path will begin at the main routine. The path will be terminated when either the end of the main routine or a "STOP" statement in any routine is encountered.

[2] START x

The command indicates that execution of the path will begin at the routine named "x". The path will be terminated when either

a "RETURN" statement in routine "x" or a "STOP" statement in any routine is encountered.

4. Address Declarations

An address declaration is used to indicate a particular address within the current scope at which a sequence of actions will be performed. The declaration is denoted by one of the keywords "BEFORE" or "AFTER" followed by a list of graph elements or one of the keywords "INITIAL" or "FINAL". All succeeding addressable commands (until the next address, or scope declaration) will be invoked only at the specified address. If the optional "NO" keyword is used, the succeeding addressable commands will be ignored.

The syntax of an address declaration is:

<address_declaration> := [NO] <address> {, <address>}

<address> := <scope_address> | <graph_address>

Each type of address (scope and graph address) will be explained separately.

4.1. Scope Address

A scope address is a special address used to indicate that the succeeding addressable commands should be performed prior to ("INITIAL") or following ("FINAL") processing the scope specified in the preceding scope declaration. Scope addresses are used to perform initialization before processing the scope or to determine the status of various quantities after processing the scope.

The syntax of a scope address is:

<scope_address> := INITIAL | FINAL

The following are examples of scope addresses.

[1] INITIAL

The address indicates that the succeeding addressable commands will be performed prior to the processing of the current scope.

[2] FINAL

The address indicates that the succeeding addressable commands will be performed following the processing of the current scope.

4.2. Graph Address

A graph address is used to indicate that the succeeding addressable commands should be performed at a specific point in a routine. A position keyword is used to indicate whether the commands should be performed prior to ("BEFORE") or following ("AFTER") the indicated address. A graph element is used to specify a unique element of a routines control flow graph. The graph is an internal representation of the routine in which statements are represented as nodes and possible flow of control between two statements as edges. Similarly, a graph node is denoted by a single statement designator while a graph edge is denoted by two statement designators (which describe the two nodes with an edge in common).

The syntax of a graph address is:

<graph_address> := <position> <graph_element> {, <graph_element>}

<graph_element> := <graph_edge> | <graph_group> | <graph_node>

<graph_group> := (<graph_edge>)

<graph_edge> := <graph_node> - <graph_node>

<graph_node> := <statement_designator>

<position> := BEFORE | AFTER

In the following examples of graph addresses, "s" and "t" are used to represent any statement designator.

[1] BEFORE s

The address indicates that the succeeding addressable commands will be performed prior to the graph node identified by "s".

[2] AFTER s

The address indicates that the succeeding addressable commands will be performed following the graph node identified by "s".

[3] AFTER s , t

The address indicates that the succeeding addressable commands will be performed following the graph node identified by "s" and following the graph node identified by "t".

[4] AFTER s - t

The address indicates that the succeeding addressable commands will be performed following the graph edge which connects the graph node identified by "s" and the graph node identified by "t".

5. Statement Designators

A statement designator is used to specify a particular statement in the current scope. The statement designator may identify a unique statement in a routine (statement label or relative location) or any number of statements (text occurence). The delimiter symbol "/" is used to indicate that a combination of two (or more) statement classes must appear in the same statement for that statement to be a designated statement.

The syntax of a statement designator is:

```
<statement_designator> := <statement_group> | <statement_list>
<statement_group> := ( <statement_list> )
<statement_list> := <statement_class> {/ <statement_class>}
<statement_class> := <statement_label> |
                     <ralative_location> |
                     <statement_type> |
                     <identifier_context>
```

Each type of statement class will be explained separately.

5.1. Statement Label

A statement label designator is used to specify a statement in the current scope. Any FORTRAN statement in the scope which is prefixed with the specified label is a designated statement.

The syntax of a statement label designator is:

<statement_label> := LABEL <label_phrase>

<label_phrase> := <label> | <label_group>

<label_group> := (<label> {, <label>})

<label> := * | integer

In the following examples of statement label designators, "m" and "n" are used to represent any FORTRAN label ("1" to "99999").

[1] LABEL *

The command indicates that any statement in the current scope prefixed with a FORTRAN label is a designated statement.

[2] LABEL m

The command indicates that any statement in the current scope prefixed with the FORTRAN label "m" is a designated statement.

[3] LABEL (m , n)

The command indicates that any statement in the current scope prefixed with either the FORTRAN label "m" or "n" is a designated statement.

5.2. Relative Location

A relative location designator is used to specify a statement in the current scope. All executable statements in a routine are assigned an integer relative location which indicates their location relative to the beginning of that routine. A statement may be designated by refering to this relative location index.

The syntax of a relative location designator is:

<relative_location> := BLOCK <block_phrase>

<block_phrase> := <block_name> | <block_group>

<block_group> := (<block_name> | {, <block_name>})

<block_name> := * | integer

In the following examples of block number designators, "i" and "j" are used to represent any positive integer.

[1] BLOCK *

The command indicates that every statement in the current scope is a designated statement.

[2] BLOCK i

The command indicates that the "i"-th statement relative to the beginning of the current scope is a designated statement.

[3] BLOCK i , j

The command indicates that the "i"-th and "j"-th statements relative to the beginning of the current scope are designated statements.

5.3. Statement Type

A statement type designator is used to specify a particular type of FORTRAN statement in the current scope. Any number of statements may be designated with a single statement type designator.

The syntax of a statement type designator is:

<statement_type> := STOP | GO_TO | RETURN | CONTINUE

The following are examples of statement type designators.

[1] STOP

The command indicates that any "STOP" statement in the current scope is a designated statement.

[2] GO_TO

The command indicates that any "GOTO" statement (including any computed "GOTO") in the current scope is a designated statement.

[3] RETURN

The command indicates that any "RETURN" statement in the current scope is a designated statement.

[4] CONTINUE

The command indicates that any "CONTINUE" statement in the current scope is a designated statement.

5.4. Variable Context

An variable context designator is used to specify a particular type of FORTRAN statement which contains a reference to a specified variable or routine name. The specified variable may be given a value by the statement (definition usage) or require a value for the statement to be executed (reference usage). The delimiter symbol "/" is used to

[2] ASSIGNMENT v

The command indicates that any statement in the current routine defining the variable "v" on the left hand side of an assignment statement is a designated statement.

[3] DO_INDEX v

The command indicates that any statement in the current routine defining the variable "v" as the control variable of a DO statement is a designated statement.

[4] PARAMETER v

The command indicates that any statement in the current routinee defining the variable "v" as a formal parameter in a routine header (SUBROUTINE or FUNCTION) is a designated statement.

[5] HEADER v

The command indicates that any statement in the current routine defining the variable as the name of a routine in a routine header (SUBROUTINE or FUNCTION) is a designated statement.

[6] DEFINITION v

The command indicates that any statement in the current routine defining the variable "v" is a designated statement. A defining statement satisfies one of the above.

5.4.2. Reference Usage Context

A reference usage context designator is used to specify a particular

type of FORTRAN statement which contains a specified variable in a referencing usage. A referencing statement is one which requires a value of the specified variable.

The syntax of a reference usage context designator is:

<reference_usage> := OUTPUT | ARITHMETIC | DO_PARAMETER |
ARGUMENT | EXTERNAL | CONDITIONAL |
SUBSCRIPT | REFERENCE

In the following examples of definition usage context designators, "v" is used to represent any variable.

[1] OUTPUT v

The command indicates that any statement in the current routine referencing the variable "v" as the object of a WRITE statement is a designated statement.

[2] ARITHMETIC v

The command indicates that any statement in the current routine referencing the variable "v" in an arithmetic expression on the right hand side of an assignment statement is a designated statement.

[3] DO_PARAMETER v

The command indicates that any statement in the current routine as an initial, terminal, or incremental variable value in a DO statement is a designated statement.

[4] ARGUMENT v

The command indicates that any statement in the current routine referencing the variable "v" in an argument expression of a routine invocation (SUBROUTINE or FUNCTION) is a designated statement.

[5] EXTERNAL v

The command indicates that any statement in the current routine referencing the variable "v" as the object of a routine invocation (SUBROUTINE or FUNCTION) is a designated statement.

[6] CONDITIONAL v

The command indicates that any statement in the current routine referencing the variable "v" in the conditional expression of an IF statement is a designated statement.

[7] SUBSCRIPT v

The command indicates that any statement in the current routine referencing the variable "v" in the subscript expression of an array reference is a designated statement.

[8] REFERENCE v

The command indicates that any statement in the current routine referencing the variable "v" is a designated statement. A referencing statement satisfies one of the above.

6. Addressable Commands

An addressable command is addressed to a particular graph or scope address by the preceeding address declaration. An addressable command must be addressed to a particular point - it modifies or controls local events at that address. The command is denoted by one of the keywords "DISPLAY", "ASSERT", "ASSUME", "ASSIGN", "BREAK", "STOP", or "RETURN" and may be followed by information specific to the particular command.

The syntax of an addressable command is:

```
<addressable_command> := <display_command> |
                         <predicate_command> |
                         <assign_command> |
                         <control_command>
```

Each type of addressable command will be explained separately.

6.1. Display Commands

A display command is used to display information about an action which takes place in the current scope. The information will be displayed when the specified address is encountered. If the optional "NO" keyword is used, the command will be ignored.

The syntax of a display command is:

```
<display_command> := [NO] DISPLAY <display_list>
<display_list> := <display_category> {, <display_category>}
```

<display_category> := <path_status> |
<variable_evaluation> |
<description_string>

Each type of display category will be explained separately.

In the following examples of display commands, "a" and "b" are used to represent any addressable category.

[1] DISPLAY a

The command indicates that information of the category "a" will be displayed at the specified address.

[2] DISPLAY a , b

The command indicates that information of both categories "a" and "b" will be displayed at the specified address.

6.1.1. Path Status

A path status display command is used to check the status of the path which has been generated to reach the specified address. Three types of path status may be checked. A path condition command is used to display the path condition for the current path. A path solution command is used to display a solution (if any) to the path condition. A path description command is used to display the particular sequence of graph elements which compose the path.

A control flow phrase is used to indicate the degree of refinement to be used in displaying the path for a "PATH DESCRIPTION" command. The

phrase "BY EXTERNALS" indicates that only transfers to external routines will be used to describe the path. The phrase "BY BRANCHES" indicates that only non-sequential transfers of control will be used to describe the path. The phrase "BY BLOCKS" indicates that all transfers of control (including sequential) will be used to describe the path. If the control flow phrase is ommitted, the default phrase "BY BLOCKS" is assumed.

The syntax of a path status display command is:

<path_status> := PATH <status_value>

<status_value> := CONDITION | SOLUTION | <description>

<description> := DESCRIPTION [<control_flow_phrase>]

<control_flow_phrase> := BY <control_flow>

<control_flow> := EXTERNALS | BRANCHES | BLOCKS

In the following examples of path status display commands, "c" is used to represent any control flow keyword ("EXTERNALS", "BRANCHES", or "BLOCKS").

[1] DISPLAY PATH CONDITION

The display category indicates that the current path condition will be displayed at the specified address.

[2] DISPLAY PATH SOLUTION

The display category indicates that a solution (if any) to the path condition of the current path will be displayed at the specified address.

[3] DISPLAY PATH DESCRIPTION

The display category indicates that the sequence of block numbers which compose the current path will be displayed at the specified address.

[4] DISPLAY PATH DESCRIPTION BY c

The display command indicates that the sequence of control flow transfers of type "c" ("EXTERNALS", "BRANCHES", or "BLOCKS") which compose the current path will be displayed at the specified address.

6.1.2. Variable Evaluation

A variable evaluation display command is used to display a particular aspect of a variable. An aspect keyword is used to specify whether the symbolic value ("VALUE") or current range ("RANGE") of the specified variables should be displayed at the specified address.

The syntax of a variable evaluation display command is:

<variable_evaluation> := <variable_aspect> <variable_phrase>
<variable_aspect> := VALUE | RANGE
<variable_phrase> := <variable_name> | <variable_group>
<variable_group> := (<variable_name> {, <variable_name>})
<variable_name> := * | identifier

In the following examples of variable evaluation display commands, "a" is used to represent an variable aspect keyword ("VALUE" or "RANGE").

and "v" and "w" are used to represent any variable name.

[1] DISPLAY a *

The display category indicates that the aspect "a" of all variables in the current scope will be displayed at the specified address.

[2] DISPLAY a v

The display category indicates that the aspect "a" of the variable "v" in the current scope will be displayed at the specified address.

[3] DISPLAY a (v , w)

The display category indicates that the aspect "a" of both variables "v" and "w" in the current scope will be displayed at the specified address.

6.1.3. Description String

A description string display command is used to display a symbolic string. The string to be displayed is surrounded by the special character "$". If the character "$" is to be displayed, the character must appear twice within the string (ie. the string "$PRINT $$ HERE$", will be displayed as "PRINT $ HERE").

The syntax of a description string display command is:

<description_string> := STRING <symbolic_string>

<symbolic_string> := $ string $

The following is an example of a description string display command.

[1] DISPLAY STRING $ string $

The command indicates that the symbolic value "string" will be displayed at the specified address.

6.2. Predicate Commands

A predicate command is used to test or modify the current path condition. The command is denoted by one of the keywords "ASSERT", "ASSUME" or "WHEN" followed by information specific to that command.

The syntax of a predicate command is:

<predicate_command> := <assert_command> |
<assume_command> |
<when_command>

Each type of predicate command will be explained separately.

The logical expression contained in the particular predicate command denoted will be tested against the current path condition and will generate one of the following results.

[1] INCONSISTENT

The result indicates that the predicate is not satisfied by any input value which would execute the current path.

[2] CONSISTENT

The result indicates that the predicate is satisfied by at least

one of the input values which would execute the current path.

[3] VALID

The result indicates that the predicate is satisfied by all of the input values which would execute the current path.

6.2.1. Assert Command

An assert command is used to verify that a particular relationship holds among a given set of variables at the specified address. If the assertion is not "VALID" and the optional "OR ABORT" phrase is used, the current path will be terminated (otherwise, the path will continue). If the optional "NO" keyword is used, the command will be ignored.

The syntax of an assert command is:

<assert_command> := [NO] ASSERT (<logical_expression>) [OR ABORT]

In the following examples of assert commands, "1" is used to represent any logical expression.

[1] ASSERT (1)

The command indicates that the predicate expression "1" must be valid. The expression will be tested against the current path condition and a message will be displayed reflecting the result of the test.

[2] ASSERT (1) OR ABORT

The command indicates that the predicate expression "1" must be

valid. The expression will be tested against the current path condition and a message will be displayed reflecting the result of the test. If the result is not "VALID", the current path will be terminated. Otherwise, the path will continue.

6.2.2. Assume Command

An assume command is used to place constraints on the values which an input variable Eceive. The constraints will be compared to the current path condition. If the assumption is "INCONSISTENT", the current path will be terminated. Otherwise, the predicate will be added to the current path condition and the path will continue. If the optional "NO" keyword is used, the command will be ignored.

The syntax of an assume command is:

<assume_command> := [NO] ASSUME (<logical_expression>)

In the following examples of assume commands, "l" is used to represent any predicate expression.

[1] ASSUME (1)

The command indicates that the predicate expression "l" will be compared to the current path condition. If the result is "INCONSISTENT", the current path will be terminated. Otherwise, the predicate will be added to the current path condition.

6.2.3. When Command

A when command is used to limit execution of a group of addressable commands to when a given condition is satisfied. The predicates will be compared (in the order that they occur) to the current path condition. When the first predicate is satisfied (not "INCONSISTENT") the predicate will be added to the path condition and the corresponding group of addressable commands will be performed. All other predicates and addressable command will be ignored. If the optional "OTHERWISE" phrase is used and all predicates are "INCONSISTENT", the group of addressable commands in the otherwise phrase will be preformed. If the optional "NO" keyword is used, the entire command will be ignored.

The syntax of a when command is:

<when_command> := [NO] <when_clause> [<alternate_group>] END

<when_clause> := WHEN (<logical_expression>) [<addressable_group>]

<alternate_group> := {<else_clause>} [<otherwise_clause>]

<else_clause> := ELSE <when_clause>

<otherwise_clause> := OTHERWISE [<addressable_group>]

<addressable_grouP> := <addressable_command> {<addressable_command>}

In the following examples of when commands, "1" and "k" are used to represent any predicate expression, and "a", "b", "c", and "d" are used to represent any addressable command.

[1] WHEN (1) a b END

The command indicates that the predicate "1" will be compared to the current path condition. If the predicate is satisfied (not "INCONSISTENT"), the predicate will be added to the current path

condition, and the addressable commands "a" and "b" will be performed.

[2] WHEN (l) a b ELSE WHEN (k) c d END

The command indicates that the predicates "l" and "k" will be compared to the current path condition. The first predicate that is satisfied (not "INCONSISTENT") will be added to the current path condition, and the corresponding group of addressable commands ("a" and "b", or "c" and "d") will be performed.

[3] WHEN (l) a b OTHERWISE c d END

The command indicates that the predicate "l" will be compared to the current path condition. If the predicate is satisfied (not "INCONSISTENT"), the predicate will be added to the current path condition, and the addressable commands "a" and "b" will be performed. If the predicate is "INCONSISTENT", the addressable commands "c" and "d" will be performed.

6.3. Assign Command

An assign command is used to assign a value to a particular variable at the specified address. The value may be either a symbolic value or a value based on an expression of variables and/or constants. The particular variable may be an array or partial array described with an implied do loop. Every element described by the implied do loop will be assigned the specified value. If the optional "INDEX" keyword is used,

each element in the array will be assigned a unique value (based on the specified value). If the optional "NO" keyword is used, the command will be ignored.

The syntax of a assign command is:

<assign_command> := [NO] ASSIGN <element> = <assign_value>
<element> := identifier | <implied_do>
<assign_value> := arithmetic_expression | <symbolic_value>
<symbolic_value> := <symbolic_string> [INDEX]
<symbolic_string> := $ string $
<implied_do> := (<element> , <do_control>)
<do_control> := identifier = <do_range>
<do_range> := <do_value> , <do_value> [, <do_value>]
<do_value> := identifier | integer

In the following examples of assign commands, "v" is used to represent any variable name, and "e" is used to represent any FORTRAN arithmetic expression.

[1] ASSIGN v = e

The command indicates that the value of the arithmetic expression "e" will be assigned to the variable "v". If "v" is an array, each element in the array will be assigned the value of the expression "e".

[2] ASSIGN v = $ string $

The command indicates that the symbolic value "string" will be assigned to the variable "v". If "v" is an array, each element

in the array will be assigned the symbolic value "string".

[3] ASSIGN v = $ string $ INDEX

The command indicates that the symbolic value "string" will be assigned to the variable "v". If "v" is an array, each element in the array will be assigned the symbolic value "string(j)" where "j" is the constant corresponding to the particular index of that element.

6.4. Control Commands

A control command is used to control processing of the current path. The action taken depends upon the particular control command encountered. If the current session is batch oriented, a "BREAK" command will be ignored. If the optional "NO" keyword is used, the command will be ignored.

The syntax of an addressable control command is:

<control_command> := [NO] <control_action>

<control_action> := <unit_control> |
<path_control>

Each type of control command will be explained separately.

6.4.1. Path Control Commands

A path control command is used to control processing of the current

path. Processing may be temporarily suspended in order to enter new AID commands by using the "BREAK" path control command. Processing will continue when the "RESUME" interactive control command is entered.

The syntax of a path control command is:

<path_control> := STOP | RETURN | BREAK

The following are examples of path control commands:

[1] STOP

The command indicates that the current path will be terminated. (Note that a FORTRAN STOP statement has the same effect.)

[2] RETURN

The command indicates that the current routine will return control to its calling routine. (Note that a FORTRAN RETURN statement has the same effect.)

[3] BREAK

The command indicates that the current path will be temporarily suspended when the command is encountered if the current session is interactive. The path will continue when the interactive control command "RESUME" is entered. If the current session is batch oriented, the command will be ignored.

6.4.2. Unit Control Command

A unit control command is used to position the symbolic unit to the

particular component which will be used during the processing of the current path. If the unit control command is ommitted, the symbolic unit will be automatically positioned to the beginning of the next component (or the beginning of the first component, if none is found) prior to the processing of the current path. If the optional "NO" keyword is used, the command will be ignored.

The syntax of a unit control command is:

<unit_control> := [NO] POSITION <unit_list> TO <component>

<unit_list> := UNIT <unit_phrase>

<unit_phrase> := <unit_name> | <unit_group>

<unit_group> := (<unit_name> {, <unit_name>})

<unit_name> := * | <unit_value>

<unit_value> := identifier | integer

<component> := * | integer

In the following examples of unit control commands, "v" and "w" are used to represent any FORTRAN unit name, "i" is used to represent any FORTRAN unit number, and "n" is used to represent any integer.

[1] POSITION UNIT * TO n

The command indicates that all symbolic units will be set to the beginning of the component numbered "n" before processing the current path.

[2] POSITION UNIT v TO n

The command indicates that the symbolic unit "v" will be set to the beginning of component number "n" before processing the

current path.

[3] POSITION UNIT i TO n

The command indicates that the symbolic unit "i" will be set to the beginning of component number "n" before the processing of the current path.

[4] POSITION UNIT (v , w) TO n

The command indicates that the symbolic units "v" and "w" will be set to the beginning of component number "n" (on both units) before the processing of the current path.

7. Interactive Commands

An interactive command is used during an interactive session to indicate that the current path should either begin processing ("PROCESS") or continue processing ("RESUME"). If an interactive command is entered during a batch oriented session, the command will be ignored. If the optional "NO" keyword is used, the command will be ignored.

The syntax of an interactive command is:

```
<interactive_command> := [NO] <interactive_action>
<interactive_action> := <process_path> |
                        <resume_path>
<process_path> := PROCESS <range_value>
<resume_path> := RESUME
<range_value> := * | <block>
```

In the following examples of interactive commands, "i" is used to represent any integer.

[1] PROCESS *

The command indicates that processing of the lowest numbered path not yet processed will begin. Processing of paths will continue until all paths have been processed.

[2] PROCESS i

The command indicates that processing of the lowest numbered path not yet processed will begin. Processing will continue until "i" paths have been processed.

[3] RESUME

The command indicates that processing of the current path will resume at the point that the addressable command "BREAK" was encountered. If the current session is batch oriented, the command will be ignored.

8. Help Facility

The help facility is used to request information about the syntax or semantics of an AID command. The help facility utilizes a graph-like structure which reflects the BNF syntax description of AID. A sequence of help commands is used to specify what portion of the help graph will be traversed during that help session. If the optional help value phrase is used, traversal of the graph will begin at the specified node of the help graph. Otherwise, the root node of the syntax graph will be the beginning node for the traversal. In addition, a short description of the use of the help facility will be presented. If the optional "NO" keyword is used, the help session will be ignored.

The syntax of a help session is:

<help_session> := [NO] HELP [<help_value>] {<help_command>} END

<help_value> := (<node_value>)

<node_value> := * | <node_name>

Each type of help command will be explained separately.

In the following descriptions of help sessions, "n" is used to represent any node name, and "p" and "q" are used to represent any help command.

[1] HELP p END

The session indicates that the help facility will be invoked at the root node of the help graph. A short description of the use of the help facility will be displayed. The help command "p" will be performed and the help session will be terminated.

[2] HELP p q END

The session indicates that the help facility will be invoked at the root node of the help graph. A short description of the use of the help facility will be displayed. The help commands "p" and "q" will be performed and the help session will be terminated.

[3] HELP (*) p END

The session indicates that the help facility will be invoked at the node named "*" (root node) the help graph. The help command "p" will be performed and the help session will be terminated.

[4] HELP (n) p END

The session indicates that the help facility will be invoked at the node named "n" of the help graph. The help command "p" will be performed and the help session will be terminated.

8.1. Help Commands

A help command is used to request information about the syntax or semantics of an AID command or to traverse the help graph. A help command can be used to traverse the graph or to gather information about the nature of the graph. If the optional node number phrase is omitted from a graph traversal command, the action taken will depend on the type of the current session. If the ATTEST session is interactive, a numbered list of the direct descendants will be displayed and a request will be made for the number of the new current node. If the session is

batch oriented, the first direct descendant which has not been visited in this help session will become the new current node.

The syntax of a help command is:

```
<help_command> := <traversal_command> |
                  <position_command> |
                  <information_command>
<traversal_command> := <direction> [<node_number>]
<direction> := SUCCESSOR | PREDECESSOR
<position_command> := FIND <node_value>
<information_command> := EXPLAIN
<node_value> := * | <node_name>
<node_number> := integer
```

In the following examples of help commands, "n" is used to represent any node name.

[1] SUCCESSOR

The command indicates that a direct descendant of the current node will become the new current node. If more than one direct descendant exists, the action taken will depend on the type of the current session as described above.

[2] SUCCESSOR i

The command indicates that the direct descendant numbered "i" of the current node will become the new current node.

[3] PREDECESSOR

The command indicates that a direct ancestor of the current node will become the new current node. If more than one direct ancestor exists, the action taken will depend on the type of the current session as described above.

[4] PREDECESSOR i

The command indicates that the direct ancestor numbered "i" of the current node will become the new current node.

[5] FIND *

The command indicates that the node named "*" (root node) of the help graph will become the new current node.

[6] FIND n

The command indicates that the node named "n" of the help graph will become the new current node. (Note that the named node may be any node in the help graph - not just the direct relative of the current node.)

[7] EXPLAIN

The command indicates that the semantic definition contained in the current node of the help graph will be displayed.

Appendix A. AID Syntax Summary

The following is a complete definition of the ATTEST Interface Description language. The start symbol for the BNF grammar is the syntactic category "interface description".

<scope_declaration> := BEGIN [AID] {<path_unit>} END [AID]

<path_unit> := [<path_declaration>] {<routine_unit>}

<routine_unit> := [<routine_declaration>] {<entry_unit>}

<entry_unit> := [<entry_declaration>] <command_unit>

<command_unit> := [<procedural_group>] {<address_unit>}

<procedural_group> := <procedural_command> {<procedural_command>}

<address_unit> := <address_declaration> {<addressable_command>}

<path_declaration> := [NO] PATH <path_list>

<path_list> := <path_scope> {, <path_scope>}

<path_scope> := * | <path_name>

<path_name> := integer [.. <path_range>]

<path_range> := * | integer

<routine_declaration> := [NO] ROUTINE <routine_list>

<routine_list> := <routine_scope> {, <routine_scope>}

<routine_scope> := * | <routine_name>

<routine_name> := identifier [.. <routine_limit>]

<routine_limit> := identifier | <routine_range>

<routine_range> := * | integer

<entry_declaration> := [NO] ENTRY <entry_list>

```
<entry_list> := <entry_scope> {, <entry_scope>}
<entry_scope> := * | <entry_name>
<entry_name> := integer [.. <entry_range>]
<entry_range> := * | integer

<procedural_command> := <trace_command> |
                        <weight_command> |
                        <connect_command> |
                        <attempt_command> |
                        <start_command>

<trace_command> := [NO] TRACE <trace_list>
<trace_list> := <trace_category> {, <trace_category>}
<trace_category> := <control_flow> |
                    <constraint_flow> |
                    <external_data_flow>

<control_flow> := EXTERNALS | BRANCHES | BLOCKS

<constraint_flow> := CONSTRAINTS

<data_flow> := <external_data_flow> |
      <internal_data_flow> |
                        <unit_data_flow>

<external_data_flow> := <external_type> <external_phrase>
<external_type> := PARAMETER | GLOBAL
<external_phrase> := <variable_name> | <external_group>
<external_group> := ( <variable_name> {, <variable_name>} )
```

<variable_name> := * | identifier

<local_data_flow> := LOCAL <local_phrase>

<local_phrase> := <local_name> | <local_group>

<local_group> := (<local_name> {, <local_name>})

<local_name> := * | variable

<unit_data_flow> := UNIT <unit_phrase>

<unit_phrase> := <unit_name> | <unit_group>

<unit_group> := (<unit_name> {, <unit_name>})

<unit_name> := * | <unit_value>

<unit_value> := identifier | integer

<weight_command> := [NO] WEIGHT <weight_list> WITH <weight_value>

<weight_list> := <graph_element> {, <graph_element>}

<graph_element> := <graph_edge> | <graph_group> | <graph_node>

<graph_group> := (<graph_edge>)

<graph_edge> := <graph_node> - <graph_node>

<graph_node> := <statement_designator>

<weight_value> := [-] integer

<connect_command> := [NO] CONNECT <connect_list> [OR ABORT]

<connect_list> := <path_component> {, <path_component>}

<path_connect> := <graph_element> | <repeat_group>

<graph_element> := <graph_edge> | <graph_group> | <graph_node>

<graph_group> := (<graph_edge>)

<graph_edge> := <graph_node> - <graph_node>

<graph_node> := <statement_designator>

```
<repeat_group> := <repeat_value> ( <component_list> )
<repeat_value> := * | <repeat_scope>
<repeat_scope> := integer [.. <repeat_limit>]
<repeat_limit> := * | integer

<attempt_command> := [NO] ATTEMPT <attempt_limit> TIMES
<attempt_limit> := * | integer

<start_command> := [NO] START <start_location>
<start_location> := * | identifier

<address_declaration> := [NO] <address> {, <address>}
<address> := <scope_address> | <graph_address>

<scope_address> := INITIAL | FINAL

<graph_address> := <position> <graph_element> {, <graph_element>}
<graph_element> := <graph_edge> | <graph_group> | <graph_node>
<graph_group> := ( <graph_edge> )
<graph_edge> := <graph_node> - <graph_node>
<graph_node> := <statement_designator>
<position> := BEFORE | AFTER

<statement_designator> := <statement_group> | <statement_list>
<statement_group> := ( <statement_list> )
<statement_list> := <statement_class> {/ <statement_class>}
<statement_class> := <statement_label> |
                     <ralative_location> |
                     <statement_type> |
```

<identifier_context>

<statement_label> := LABEL <label_phrase>

<label_phrase> := <label> | <label_group>

<label_group> := (<label> {, <label>})

<label> := * | integer

<relative_location> := BLOCK <block_phrase>

<block_phrase> := <block_name> | <block_group>

<block_group> := (<block_name> | {, <block_name>})

<block_name> := * | integer

<statement_type> := STOP | GO_TO | RETURN | CONTINUE

<variable_context> := <variable_usage> <variable_phrase>

<variable_usage> := <definition_usage> | <reference_usage>

<variable_phrase> := * | <variable_name> | <variable_group>

<variable_group> := (<variable_term> {, <variable_term>})

<variable_term> := <variable_name> {/ <variable_name>}

<variable_name> := * | variable

<definition_usage> := INPUT | ASSIGNMENT | DO_INDEX |
PARAMETER | HEADER | DEFINITION

<reference_usage> := OUTPUT | ARITHMETIC | DO_PARAMETER |
ARGUMENT | EXTERNAL | CONDITIONAL |
SUBSCRIPT | REFERENCE

<addressable_command> := <display_command> |
<predicate_command> |

<assign_command> |

<control_command>

<display_command> := [NO] DISPLAY <display_list>

<display_list> := <display_category> {, <display_category>}

<display_category> := <path_status> |

<variable_evaluation> |

<description_string>

<path_status> := PATH <status_value>

<status_value> := CONDITION | SOLUTION | <description>

<description> := DESCRIPTION [<control_flow_phrase>]

<control_flow_phrase> := BY <control_flow>

<control_flow> := EXTERNALS | BRANCHES | BLOCKS

<variable_evaluation> := <variable_aspect> <variable_phrase>

<variable_aspect> := VALUE | RANGE

<variable_phrase> := <variable_name> | <variable_group>

<variable_group> := (<variable_name> {, <variable_name>})

<variable_name> := * | identifier

<description_string> := STRING <symbolic_string>

<symbolic_string> := $ string $

<predicate_command> := <assert_command> |

<assume_command> |

<when_command>

<assert_command> := [NO] ASSERT (<logical_expression>) [OR ABORT]

```
<assume_command> := [NO] ASSUME ( <logical_expression> )

<when_command> := [NO] <when_clause> [<alternate_group>] END
<when_clause> := WHEN ( <logical_expression> ) [<addressable_group>]
<alternate_group> := {<else_clause>} [<otherwise_clause>]
<else_clause> := ELSE <when_clause>
<otherwise_clause> := OTHERWISE [<addressable_group>]
<addressable_grouP> := <addressable_command> {<addressable_command>}

<assign_command> := [NO] ASSIGN <element> = <assign_value>
<element> := identifier | <implied_do>
<assign_value> := arithmetic_expression | <symbolic_value>
<symbolic_value> := <symbolic_string> [INDEX]
<symbolic_string> := $ string $
<implied_do> := ( <element> , <do_control> )
<do_control> := identifier = <do_range>
<do_range> := <do_value> , <do_value> [, <do_value>]
<do_value> := identifier | integer

<control_command> := [NO] <control_action>
<control_action> := <unit_control> |
                    <path_control>

<path_control> := STOP | RETURN | BREAK

<unit_control> := [NO] POSITION <unit_list> TO <component>
<unit_list> := UNIT <unit_phrase>
<unit_phrase> := <unit_name> | <unit_group>
```

```
<unit_group> := ( <unit_name> {, <unit_name>} )
<unit_name> := * | <unit_value>
<unit_value> := identifier | integer
<component> := * | integer

<interactive_command> := [NO] <interactive_action>
<interactive_action> := <process_path> |
                        <resume_path>
<process_path> := PROCESS <range_value>
<resume_path> := RESUME
<range_value> := * | <block>

<help_session> := [NO] HELP [<help_value>] {<help_command>} END
<help_value> := ( <node_value> )
<node_value> := * | <node_name>

<help_command> := <traversal_command> |
                  <position_command> |
                  <information_command>
<traversal_command> := <direction> [<node_number>]
<direction> := SUCCESSOR | PREDECESSOR
<position_command> := FIND <node_value>
<information_command> := EXPLAIN
<node_value> := * | <node_name>
<node_number> := integer
```