SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 115–129 (1978)

Event Associations in SNOBOL4 for Program Debugging

DAVID R. HANSON

Department of Computer Science, Yale University, 10 Hillhouse Avenue, New Haven, Connecticut 06520, U.S.A.

SUMMARY

An event association facility for the SNOBOL4 programming language is described. This facility permits the execution of a programmer-defined function to be associated with the occurrence of a specified event. The events with which associations can be made are those applicable to program debugging. Associations can be made with events such as variable referencing, statement execution, program interruption, function call and return, and execution-time errors. By making event associations available at the source-language level, debugging aids can be written in SNOBOL4 itself, using the full capabilities of that language. As illustrated by several examples, this approach facilitates the implementation of simple yet powerful debugging aids written in the same language as the programs to be debugged. Event associations provide a mechanism for the unification of the existing SNOBOL4 debugging facilities, and a basis for the addition of other events. The implementation of event associations is also described.

KEY WORDS SNOBOL4 Debugging aids Programming languages Variable associations

INTRODUCTION

In a recent paper,[1] a new facility for the SNOBOL4 programming language[2] permitting the execution of a programmer-defined function to be associated with the act of referencing a variable was described. This language feature enabled such seemingly disjoint facilities as input and output associations, value tracing and keywords to be described in terms of a single mechanism. Applications of this mechanism, called variable association, include datatype coercion, data structure access and manipulation, generators, input and output processing, program monitoring and program debugging.

Referencing a variable is only one of many *events* that occur during program execution. As noted in Reference 1, a natural generalization of variable association is *event* association, in which the execution of a defined function is triggered at the occurrence of a specified event. This paper describes such a generalization of variable associations to include associations with events that facilitate program debugging.

Although events other than those helpful in program debugging might be considered, experience with variable associations indicates that the association concept is especially applicable to program debugging. In addition, better debugging tools probably have the most practical impact on SNOBOL4 users.

The event associations described in this paper unify variable associations and the existing debugging aids in SNOBOL4 into one mechanism, and provide a linguistic basis for the addition of other kinds of events.

0038–6644/78/0208–0115$01.00

Received 1 April 1977

Existing debugging facilities

SNOBOL4 contains several built-in features that aid in debugging a program. Features such as tracing and a symbolic dump facility are provided in most implementations of SNOBOL4. More recent implementations, such as SPITBOL[3] and SITBOL,[4–6] permit execution-time errors to be intercepted. The symbolic dump facility was extended in SITBOL to allow dumping of aggregates such as arrays, tables and defined data objects. All of the debugging facilities provided in SNOBOL4 are an integral part of the source language. This method gives the programmer a great deal of power with which to write debugging routines. Some programmers have complained that the debugging tools are inadequate[7] but they often overlook the fact that SNOBOL4 functions can be written for use as debugging aids.

There are several debugging features that are either missing or too costly and burdensome to implement in the source language. An example is the setting of breakpoints on any source statement. Tracing the key word &STCOUNT can be used to cause a defined function to be called before the execution of every statement, but the artifact is much too costly to be used in debugging complex programs. *Transfers* to labels can also be traced in a similar fashion. This feature provides a limited breakpoint facility but does not permit execution to be interrupted at an unlabelled statement or if a statement is flowed into.

These drawbacks are not so serious in a batch environment, but they are serious in a timesharing environment. Few high-level languages provide facilities for interactive debugging in which the interaction is in terms of the high-level language itself. Notable exceptions are the DECsystem-10 implementations of Fortran[8] and Simula.[9]

Debugging is considered by some to be a less important activity than careful program preparation using the various structured programming methodologies. Nonetheless, programmers spend countless hours debugging. This time might be substantially reduced if good debugging tools were more readily available. Good debugging tools are an essential component of a high-level language system and complement the use of structured programming methodologies. Event associations are an attempt to provide a facility that can be used to write such tools at the source-language level.

The remainder of this paper describes the event association facility, illustrates its use in a general-purpose interactive debugging program and gives a brief indication of the implementation techniques employed. Background material concerning the concept of associations is given in Reference 1. The event association facility is implemented in SITBOL,[4–6] an implementation of SNOBOL4 for the DECsystem-10.

EVENT ASSOCIATIONS

Event associations permit the programmer to intervene, via a defined function, at the occurrence of certain events during the course of program execution. In most cases, the defined function, called an association function, is called *before* the completion of the event. Failure of the association function can sometimes be used to prevent the completion of the interrupted event. For example, a variable association causes the association function to be called before assignments to the associated variable. If the association function fails, the assignment is not made. There are currently five kinds of event associations:

1. Variable associations, in which the association function is called whenever the associated variable is referenced.
2. Statement execution associations, in which the association function is called prior to the execution of the associated statement.

3. Program interruption associations, in which the association function is called when the execution of the program is interrupted by external means.
4. Function call and return associations, in which the association function is called upon entry or exit of the associated function.
5. Error associations, in which the association function is called upon the occurrence of a specified execution-time error.

Associations are made by the built-in function CONNECT:

CONNECT(*name*, *type*, *processdescription*)

The *name* is the object with which the association is made, e.g. variable name, statement, function name, etc. The *type* is a string indicating the specific event for which the association is to be made. The *processdescription* is an instance of a programmer-defined data object with at least three fields. The contents of these fields indicate the association function and the current state of the association.

More specifically, the defined datatype used for the *processdescription* can be defined by the statement

DATA("PROCESS(AFUNCT, ACTIVE, LEVEL)")

The AFUNCT field contains the name of the association function, which may be either a built-in or defined function. The contents of the ACTIVE field indicates the current status of all associations made with a particular instance of PROCESS. If the ACTIVE field contains zero, the association is considered inactive and is ignored whenever the corresponding event occurs. If the ACTIVE field contains a non-zero integer when the event occurs, the association function given in the AFUNCT field is invoked. The LEVEL field is used to indicate the position of an association with respect to existing associations in the case of multiple associations with the same event. Further details concerning its use are given in Reference 1.

When an event having an association occurs, the association function is called with three arguments:

1. The *name* of the object with which the association is made (the first argument to CONNECT).
2. An event-dependent value.
3. The process description.

The success or failure of the association function can, in some cases, be used to alter the course of subsequent execution. When an association function is called, the association that caused the function to be invoked is deactivated until the function returns. Other associations with the same event are still active, however. The following sections describe the use of the association function with each of the five kinds of event associations.

Variable associations

As described in Reference 1, variable associations permit an association function to be associated with the event of fetching the value of a variable or assigning a value to a variable. The first argument to CONNECT, *name*, is the name of the variable with which to make the association, and the second argument, *type*, is either "FETCH" or "STORE" depending for which event the association function should be called.

For a "FETCH" association, the second argument to the association function is the current value of the associated variable. If the association function fails, that reference to the variable fails. If the association function succeeds, the value returned is taken as the value of that reference to the variable. The value of the variable is not changed, however.

For a "STORE" association, the second argument to the association function is the value that is about to be assigned. Assignment is prevented if the association function fails. If the association function succeeds, the value returned is ultimately assigned to the variable.

Variable associations permit the implementation, in SNOBOL4, of SNOBOL4-style input and output, access tracing, generators and datatype coercion mechanisms. Examples are given in Reference 1.

Statement execution associations

This kind of association permits an association function to be associated with the event of statement execution. Statement execution associations are most useful for setting breakpoints at or tracing the execution of certain source statements. This event is indicated by using the string "STATEMENT" as the second argument to CONNECT.

The first argument to CONNECT indicates the statement with which to make the association. This indication is made in a machine-independent fashion by using an object of datatype CODE, which uniquely identifies a statement regardless of the statement numbering scheme used. The built-in function

WHERE(*label*, *offset*)

fails if the indicated statement does not exist. The *offset* may be positive or negative. If the statement is a valid source statement, WHERE returns an object of datatype CODE that points directly to the statement. This is identical to the kind of value returned by the built-in function CODE. In addition, since the object is of datatype CODE, a transfer can be made directly to the statement by using the direct goto in SNOBOL4.[2]

Prior to the execution of an associated statement, the association function is called with the statement number as the second argument. Upon return from the association function, the statement is executed. Success or failure of the association function and the value returned are ignored.

To illustrate the use of statement execution associations, the following association function can be used to trace the execution of the statements with which it is associated.

```
          DEFINE("STRACE(STMT, STNO, SPROCESS)") :(STRACE.END)
STRACE    OUTPUT = "executing statement" STNO
          ACTIVE(SPROCESS) = ACTIVE(SPROCESS) - 1   :(RETURN)
STRACE.END
```

The ACTIVE field of the process description is decremented by 1 each time STRACE is called. Thus the contents of the ACTIVE field serve to limit the amount of trace output on a per statement basis. To use STRACE, the appropriate process description is CONNECTed to the desired statement, e.g.

```
CONNECT(WHERE("LOOP",3), "STATEMENT",
          PROCESS("STRACE", 100, 1))
```

traces the next 100 executions of the third statement following the statement labelled LOOP.

Program interruption associations

Most timesharing systems provide a means for interrupting a running program. This capability is useful for stopping looping programs. Some systems, such as the

DECsystem-10, also provide a command that enables the interrupted program to continue from the interruption in its present state, or to continue at some specified address. While such features are of obvious utility, they are seldom used for program debugging beyond the assembly language level.

Program interruption is another of the many possible events that may occur during program execution, although its occurrence is triggered externally and cannot be determined from examination of the program. An association can be made with the event of program interruption by giving the string "INTERRUPTION" as the second argument to CONNECT. The first argument to CONNECT is ignored, but is passed to the association function when it is called.

While a statement execution association provides for interstatement control, a program interruption association allows the program to be interrupted during intrastatement execution. If the event occurs, the intermost interpreter loop is interrupted in order to call the association function. This allows intrastatement loops, such as runaway pattern matches, to be broken. If the association function succeeds, execution is continued from the point of interruption. If the function fails, the interrupted statement fails. The second argument to the association function is the statement number of the interrupted statement.

As an example of the usefulness of this kind of association, the following program fragment defines the function PEEK and associates it with the event of program interruption. If this fragment is compiled and executed with the program under test, program execution may be interrupted at any point and arbitrary SNOBOL4 expressions are evaluated in order to examine the current state of the program. When the string "GO" is entered, or end of file is signalled, PEEK returns, permitting the program to continue from the point of interruption. The built-in function EVAL evaluates SNOBOL4 expressions, and TTY is input- and output-associated with the terminal.

```
          DEFINE("PEEK(X, STNO, P)EXP")                    :(PEEK.END)
PEEK      TTY = "interrupted at stmt" STNO
PLOOP     EXP = TTY                                        :F(RETURN)
          IDENT(EXP, "GO")                                 :S(RETURN)
          TTY = EVAL(EXP)                                  :S(PLOOP)
          TTY = "failed"                                   :(PLOOP)
PEEK.END
          DATA("PROCESS(AFUNCT,ACTIVE,LEVEL)")
          CONNECT(NULL, "INTERRUPTION",
+                       PROCESS("PEEK", 1, 1))
```

Note that this rather simple yet powerful debugging aid needs to be written only once; its use requires simply prepending it to the program under test. The ability to write program fragments in the source language encourages the use of libraries containing simple debugging aids such as PEEK, and facilitates the maintenance of such libraries.

Function call and return associations

Function associations permit an association to be made with the events of function call and return. Function associations are similar to the use of defined trace functions in the existing SNOBOL4 tracing facility. The first argument to CONNECT is the name of the function with which the association is to be made, and the second argument is "CALL" or "RETURN" indicating the appropriate event.

For a function call, the association function is invoked after the arguments have been passed to the function but before transfer to the entry point of the function. The second argument to the association function is the number of the statement invoking the function. The function PEEK, used above as an association function for program interruption, can also be used as an association function for function call associations, permitting the program state to be examined when certain functions are called. For example, the statement

```
CONNECT("PARSE", "CALL", PROCESS("PEEK", 1, 1))
```

causes PEEK to be invoked on subsequent calls to PARSE. The name of the interrupted function is given by the value of X, the first argument of PEEK.

The decision to make such an association need not be made prior to program execution. If PEEK is prepended to the program under test, execution may be interrupted at any point and call associations with any number of functions can be made by entering the appropriate CONNECT expression to PEEK. A simple function, such as PEEK, can be used as the association function for many events and is often sufficient for many debugging tasks.

Another use of call associations is to localize common preprocessing of arguments in one function instead of including such processing in each function. This is especially useful to ensure that a particular argument is of the correct datatype. For example, the following association function can be used to ensure that the first argument to functions with which it is associated is an integer. The built-in function ARG(F, I) returns the name of the Ith argument of F, and CONVERT(X,T) converts X to type T failing if the conversion cannot be performed.

```
          DEFINE("INTARG(F, STNO, P)")     :(INTARG.END)
INTARG    $ARG(F,1) = CONVERT($ARG(F,1), "INTEGER")
+                                          :S(RETURN)
          OUTPUT = "bad first argument to" F
          PEEK(F, STNO, P)                 :(RETURN)
INTARG.END
```

The call to PEEK permits the argument to be changed before proceeding with the interrupted function. A slightly more general association function that handles any number of datatypes can be written by including the datatype as an additional field in the process description. The point is that the full power of SNOBOL4 is available for writing debugging aids.

For a return association, the association function is called before the arguments and the function name are restored to their previous values. Thus the first argument to the association function can be used to obtain the value returned by the function and the final values of the arguments. In addition, the value of keyword &RTNTYPE indicates the type of return (RETURN, FRETURN or NRETURN) made by the function.

Error associations

This kind of association permits an association function to be associated with the occurrence of a specific execution-time error or all such errors. Errors are indicated by a number of the form 1000*n+m, where m is a major error number (1–13) indicating the class of error, and m is a minor error number (1–999) indicating the nature of the error in detail. (A list of the possible errors is given in Reference 5.) The number of the error with which an association is to be made is given as the first argument to CONNECT, and the

string "ERROR' is given as the second argument. An error number of zero indicates the association is to be made with all possible errors.

The second argument to the association function is the number of the statement causing the error. If the association function fails, the offending statement fails and execution continues. If the association function succeeds, the offending statement is repeated under the assumption that corrective action was taken by the association function.

Removing event associations

The built-in function

DISCONNECT(*name, type, processdescription*)

disconnects event associations described by *type* and *processdescription* from *name.*

Global control of associations

Each association includes a field in the process description that indicates if the association is active. The keyword &ASSOCIATE controls the activation status of all associations. If &ASSOCIATE is zero, all associations are considered inactive. The default for &ASSOCIATE is 1, indicating that the contents of the ACTIVE field of the process description determines if the association is active.

EXPERIENCE

The inclusion of the event association facility at the source-language level facilitates the implementation of a wide range of general-purpose debugging aids, or routines that are customized for a particular set of programs. In almost all cases, event associations permit these kinds of debugging aids to be written as a separate program and compiled with the program under test. Once the program is finished, the debugging program is omitted from subsequent runs. Since the debugging programs are written in SNOBOL4, their sizes are under the programmer's control. More importantly, their use does not require modification of the program under test.

It is difficult to convince programmers to program 'defensively' and include debugging code as a part of their programs. Such advice is often met with resistance because debugging statements usually must be left in the final version of the program, and are tedious to include in the development of a program. Event associations permit a separate program to be written, perhaps only once, that is just as effective as the insertion of debugging statements.

Event associations have been used to write a number of small debugging aids such as PEEK. One of the more ambitious and useful aids is a general-purpose interactive debugger similar in spirit to the well known DDT programs for debugging assembly language programs (cf. Reference 10). This program, called SNODDT, is written in SNOBOL4 and is compiled with the program to be debugged. Variable associations are used to allow the programmer to restrict the datatype of any number of variables and intervene whenever the value of a variable is fetched or a new value stored. Statement execution associations are used to place breakpoints at any source statements. Function call and return associations are used to permit intervention at the entry to or exit from a function. A program interruption association, with SNODDT as the association function, is used so that a running program can be interrupted at any point. SNODDT is entered by typing an operating system command that causes the program to continue at a predefined address. SNODDT is also associated with the occurrence of all errors so that any execution error causes SNODDT to be called.

Besides the commands that set up associations, other commands permit the compilation and execution of SNOBOL4 code, take commands from a file and permit execution to begin at any statement.

SNODDT commands are of the general form

arg_1; arg_2; ...; $arg_n$$command

Arguments can be separated by either semicolons as shown or can be entered on separate lines. The SNODDT *command* is the one or two characters that follow the $. SNODDT commands can be given in either upper or lower case but arguments are taken in the case typed. Errors in typing a command are signalled by ?? typed after the command. The following commands are representative of the SNODDT repertoire.

The ability to compile and execute SNOBOL4 source code during execution is especially useful for debugging. The command

exp$X

executes the SNOBOL4 expression given by *exp* and types the result. For example,

A = B + C<5>$X

performs the indicated addition and assignment, and types the value assigned to A. Since a single identifier constitutes a simple expression, a command such as

FOO$X

causes the current value of FOO to be typed.

The command

code$K

passes the arguments to the SNOBOL4 CODE function for compilation. All arguments are passed to the CODE function so that multiline input is possible. For example, to define a new function F that returns the square of random number in the interval {0, 100} one can enter

```
F       F = RANDOM(100)
        F = F*F                 :(RETURN)$K
DEFINE("F()")$X
```

Notice the use of the K command to compile the function code then the subsequent use of the X command to execute the DEFINE function thereby defining the new function.

Transfer to an arbitrary statement is accomplished by the command

label; *offset*$G

The *offset* is optional and can be either positive or negative. If the command is given with no arguments, a jump to RETURN is made. This is the command that is used to begin initial execution of the program under test. Under normal circumstances, the user exits from SNODDT after an interruption of program execution by using the P command described below. If the G command is used, the function SNODDT() never returns and a portion of the system stack is lost. Since the loss of stack space is usually acceptable during debugging, the G command can be used to restart the program at any desired point.

Accesses to variables may be intercepted by using the F and S commands. These commands cause a variable association to be made with the indicated variable using SNODDT as the association function. The command

name; *exp*; *tag*$F

causes subsequent *fetches* of the value of the variable *name* to be interrupted by a call to SNODDT. The arguments *exp* and *tag* are optional. Conditional interruption can be

defined using the *exp* argument, which is a SNOBOL4 expression. At every fetch of the indicated variable, *exp* is evaluated. The program is interrupted only if *exp* succeeds. For example, suppose one wishes to monitor every fetch of variable LINENO. The command

LINENO$F

accomplishes the desired result. A message is issued whenever the value of LINENO is fetched. After several program interruptions, assume that fetches to LINENO are to be monitored only if the value of YY is less than zero. This can be done by entering

LINENO; LT(YY, 0)$F

The *tag* argument is used to construct the message issued at program interruption. This argument normally is not needed but can be used to include other information that may help identify the variable. For example, suppose PTR is used to point to the last element of a list. The command

PTR;; List pointer PTR$F

would result in a message something like

Stmt 230, List pointer PTR fetched, value = NODE #7

whenever the value of PTR was fetched.

Assignments to a variable can be intercepted by SNODDT via the command

name; *exp*; *tag*$S

The arguments *exp* and *tag* are optional and have the same use as in the F command. The S command can be used to monitor stray assignments to any variable. The act of assignment is trapped not only for the assignment operator but also for conditional and immediate value assignment during pattern matching.

After the program is interrupted by a call to SNODDT, it can be continued by the command

$P

The P command simply causes a return from SNODDT and the program proceeds from the point of interruption.

Dynamic datatype checking is initiated by the T command. The command

name; *type*; *tag*$T

restricts the values assigned to *name* to those of datatype *type*. SNODDT is entered whenever an attempt is made to assign a value of a different type to *name*. *Type* may be a programmer-defined data-type or built-in datatype such as INTEGER. Lower case types are *not* equivalent to upper case types, i.e. INTEGER is different than integer. The *tag* argument has the same use as in the F and S commands. A common error in SNOBOL4 is attempting to access the field of a value that is not a defined data object.[7] If a defined datatype NODE has been defined with two fields VALUE and LINK, and the value of PTR is expected to be a NODE, the command

PTR; NODE$T

would cause the program to be interrupted at any attempt to assign PTR a value of a different type. The P command is used to resume execution after an interruption caused by a type conflict and the assignment is performed. The value assigned can be changed to a value of the correct type during interaction with SNODDT.

The associations made by the F, S and T commands can be removed by commands such as

name$0F

The command

label; *offset*; *exp*; *tag*$B

sets a breakpoint at the source statement described by *label* and *offset*. This command uses a statement execution association. Any number of breakpoints can be set, limited only by the available memory. The arguments *exp* and *tag* are used as described above. Suppose the structure of a program is one main processing loop beginning at the statement labelled LOOP, and that the program consists of many sections each terminated by a jump back to LOOP:

```
LOOP    .
        .
                :(LOOP)
        .
        .
                :(LOOP)
        .
        .
                :(LOOP)
END
```

A useful way to monitor the execution of a program with this kind of structure is to place a breakpoint on the statement labelled LOOP and include the conditional expression TTY() as the *exp* argument. The built-in function TTY() succeeds if a line has been typed by the user and fails otherwise. Thus, the SNODDT command

LOOP; 0; TTY()$B

places a breakpoint on the statement labelled LOOP that causes program interruption only if the user types a SNODDT command. Execution is continued after a breakpoint interruption by the P command.

The command

label; *offset*$0B

removes the breakpoint from the indicated statement.

Function call and return associations are made by the commands

name; *exp*; *tag*$C
name; *exp*; *tag*$R

which cause SNODDT to be called at the entry and exit from the defined function indicated by *name*, respectively. The use of *exp* and *tag* is as described above, the P command is used to continue execution, and commands such as

name$0C

are used to remove the associations.

The facilities of SNODDT can be augmented by other associations during program execution. Additional routines and association functions that are customized for specific programs need to include only those features not provided by SNODDT. To encourage this sort of activity, the command

filename$Y

reads the indicated file containing SNODDT commands and executes those commands. The contents of *filename* can include function definitions (using the K command) and make

associations that are designed for a specific program. Thus, SNODDT provides a basis for a library of debugging aids for SNOBOL4 programs, all of which are written in SNOBOL4.

IMPLEMENTATION

As mentioned in Reference 1, a facility such as the event association facility is omitted from most languages because efficient implementation techniques are difficult to discover. This is unfortunate; well designed debugging facilities are likely to be used just as much, if not more, than the other features in a language. Such facilities are an integral part of a language and their implementation deserves the extra effort required. One advantage of providing debugging facilities as a part of the source language is that their implementation may be considerably simplified by using existing components of the system. This is the case with the event association facility, which required only an additional 200 decimal words of memory (less than 3 per cent of the system), and made use of many existing facilities.

The most important aspect of the implementation of debugging facilities is that they must have a negligible impact when not in use or in programs that do not use them. This objective was met in the implementation of event associations by ensuring that no more than a single instruction was added in order to test for the existence of an association. Details concerning the implementation of each of the five kinds of associations are given in the following sections.

Variable associations

The implementation of variable associations is described in Reference 1, and is based on the 'trapped variable' used internally in SITBOL.[4] The value of the associated variable is replaced by a trapped variable that points to a so-called trapping block containing the real value, a pointer to the process description and several other system items. Values are stored and retrieved by a single system procedure that checks for trapped variables and calls the appropriate trapping procedure. For variable associations, the trapping procedure saves the state of the SNOBOL4 virtual machine and calls the association function with the appropriate arguments. The success or failure of the association function is passed back to the function that invoked the assignment or fetch procedure for further action.

Thus, the overhead for variable associations is the one instruction that checks for the existence of a trapped variable. Note that this overhead is independent of the number of variable associations made.

Statement execution associations

Statement execution associations are made by modifying the compiled code for the affected statement. This modification is done when the association is made and is 'undone' when the association is removed. As a result, there is no overhead for statement associations unless they are used, and there is no residual overhead after an association is removed.

Compiled code in SITBOL is a Polish suffix form of the source statement with additional interpreter functions that perform operations such as statement initialization and gotos. The code for several statements is housed in a CBLOK, the code block.[4] Each statement begins with a statement header as depicted in Figure 1. The CHEAD field contains the address of the interpreter function that initiates statement execution. The CSTNO field contains the source statement number. The CNEXT and CLAST fields contain the offsets from the head of the CBLOK to the next and last statement header respectively. The

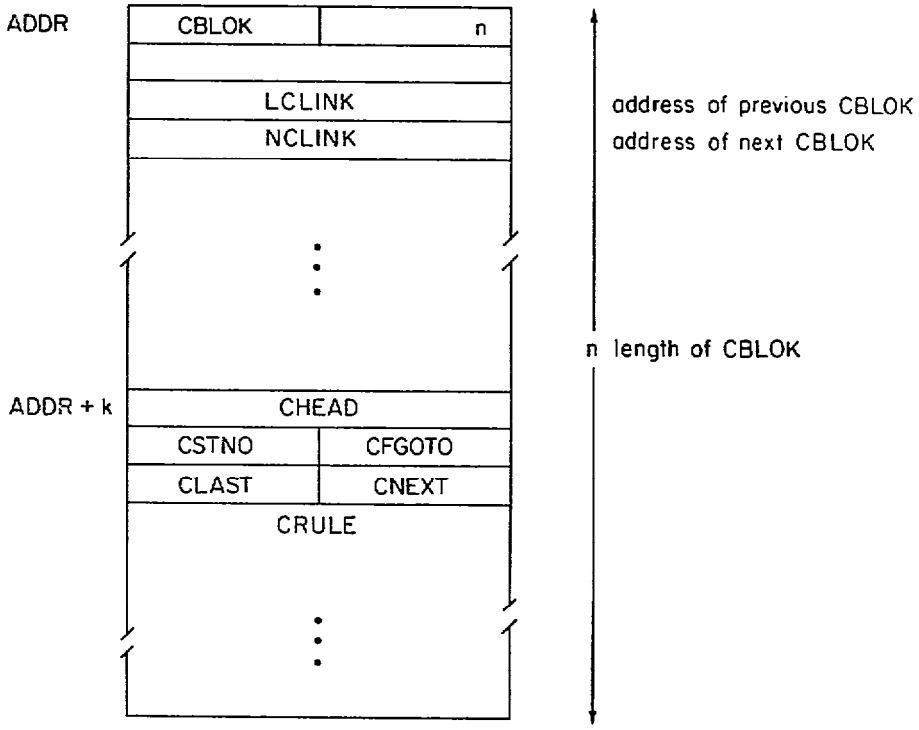

Figure 1. Code block and statement layout

CFGOTO field contains the offset of the position in the CBLOK where control should be resumed upon statement failure. The actual compiled code for the source statement begins at CRULE. A typical CBLOK contains about ten source statements. Code blocks are chained together to form a doubly-linked list. Thus the internal representation of the SNOBOL4 source program is a doubly-linked list of CBLOKs each of which contains the Polish suffix code for several statements.

The compiled code is comprised of interpreter function addresses, SNOBOL4 source language function addresses and operand addresses. The function of the interpreter is to loop through the code pushing operands onto the system stack and calling functions as they are encountered. With the exception of some interpreter functions, arguments to functions are passed on the stack, removed by the functions and results placed on the stack.

Each statement can be uniquely represented by the address of the code block housing it and the offset to the CHEAD field of the statement. A descriptor of this form is of datatype CODE and is precisely what is returned by the built-in CODE function. For example, the statement shown in Figure 1 is represented by the descriptor illustrated in Figure 2. The WHERE function returns an object of this type.

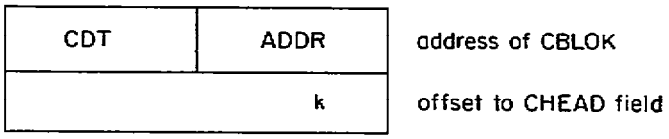

Figure 2. A descriptor for datatype CODE

Usually the CHEAD field of a statement contains the address of the interpreter function to begin a statement, BGST. When an association is made with a statement, the CHEAD field is modified to point to the breakpoint interpreter function, BRKPNT. If all associations are removed, BGST is reinstated in the CHEAD field. Only statements that have associations incur the overhead of the breakpoint interpreter function.

The process descriptions for statement associations are stored in an internal table indexed by the code descriptor for the associated statement. This table, called the breakpoint table, is allocated only if statement associations are actually used. Each entry contains a list of trapping blocks describing the associations with that statement. The trapping blocks are the same types used for variable associations and contain a pointer to the process description. The BRKPNT interpreter function locates the approriate entry in the breakpoint table, checks to see if the association is active and calls the association function. After the execution of the association function, the breakpoint function calls BGST, the begin statement function, and normal execution proceeds.

The breakpoint table is an instance of the SNOBOL4 built-in datatype TABLE, and the existing table-handling routines are used for lookup and insertion. This is a case where an existing part of the system was used, and greatly simplified the implementation of statement execution associations.

Program interruption associations

The implementation of program interruption associations proved to be the most sensitive. This is because of the requirement for program interruption during intrastatement loops. This requirement rules out a test that is performed between statements; a test must be made within the central interpreter loop.

In designing the implementation, three methods were considered. One method is to simply test a flag during every cycle through the interpreter loop. This would consist of the instruction sequence

```
SKIPE   INTFLG      ; is interrupt flag set ?
JRST    INTCMD      ; yes, perform event association
JRST    I.BUMP      ; no, proceed in interpreter loop
```

The additional overhead is the cost of the SKIPE instruction. On the KI10 central processor, this is 1·37 μs.[11] Unfortunately this must be executed even if the program is not making use of the event association facility. Since this appears in the intermost interpreter loop, the cost was judged to be too high.

A second method is to use indirect addressing in the single instruction

```
JRST  @ADDR
```

and modify the contents of ADDR to be either I.BUMP or INTCMD, the address of the event association processor. The cost of the indirection is an additional 1·02 μs on the KI10; too costly for an intermost loop.

The DEC-10 has an instruction, XCT E, that executes the instruction at address E. It also has the advantage of being one of the fastest instructions on the KI10—0·34 μs in addition to the instruction executed. Thus the method chosen was to use the instruction

```
XCT  ADDR
```

where the contents of ADDR is either

```
JRST  I.BUMP
```

to continue the interpreter loop or

```
JRST  INTCMD
```

to process the event association. This method adds a minimal amount of additional processing time to programs that do not use the facility. (A similar method is used to implement breakpoints in a Fortran debugging system.[8])

When the user interrupts the program and enters the appropriate operating system command, the contents of ADDR are changed to

```
JRST  INTCMD
```

When the XCT instruction is reached, the event processor is called. It saves the state of the virtual machine and calls the association function. Upon successful return, the state is restored, the contents of ADDR are changed to the instruction

```
JRST  I.BUMP
```

and execution continues. If the association function fails, the state is restored but the event processor causes the interrupted statement to fail.

Function call and return associations

Function associations are implemented by checking for an association at the time of call and return. This check consists of a single instruction. If a function has a call or return association, the association function is called at the appropriate time.

Error associations

Because errors are expected to occur infrequently, the efficiency of error associations is not crucial. When an error association is made, an entry is inserted into the error association table. Like the breakpoint table, this table is allocated only if error associations are used, and is a SNOBOL4 TABLE.

When an error occurs, the error number is used to index the table, the proper association (if present) is retrieved, and the association function is called. If there is no entry for the error number, a zero index is used to locate associations that should be invoked for all errors.

CONCLUSIONS

Program debugging systems are usually implemented in one of two ways. One method is to require modification of the program under test to activate various debugging aids. This is the approach most often used in debugging compilers in which additional debugging statements are introduced into the source program. Another method is to modify or monitor the running program with another program that provides the debugging aids. The second method is considered the better of the two since the modification of the text of the program under test may introduce other bugs. A disadvantage of the second method is that the debugging language is often quite different from the source language thereby requiring the user to learn two languages in order to debug a program.

The event association facility can be viewed as a contribution to either method. The programmer can modify the source text of the program under test to monitor the execution of the program. Only enough debugging code to solve the specific problems encountered needs to be added, and the additions are in the same language as the program to be debugged.

On the other hand, a completely separate program, such as SNODDT, can be written and loaded with the program under test to provide monitoring and debugging aids. The text of the original program does not need to be modified in order to use a program such as SNODDT. SNODDT is a rather elaborate example of a debugging routine. As illustrated

by PEEK, however, the most useful features can be provided by simple debugging routines. This capability is a consequence of making event associations available at the source-language level. This approach is similar to that taken in other diagnostic facilities for SNOBOL4.[12]

Event associations are machine-independent concepts and are applicable to other high-level languages. Practical use indicates that the most dynamic features of event associations, such as the ability to change the association function, are rarely used. Thus, in languages with earlier binding times than SNOBOL4, event associations could be made during compilation.

One problem with event associations is their non-uniformity. It is not sufficient to describe the general mechanism and simply enumerate the set of events with which associations can be made; each event requires a separate explanation of the arguments passed to the association function, what is done with the value returned, etc. This problem has been partially resolved in the SL5 programming language.[13] Armed with the experience gained from using event associations in SNOBOL4, a similar facility was designed for SL5. That facility, in which association functions are called filters,[14] is based on variable associations, and all events are cast in those terms. As a result, filters are more general than variable associations. For example, filters are used to describe argument binding in terms of SL5 itself. Current research is directed toward developing the filter approach used in SL5 into a uniform event association mechanism.

ACKNOWLEDGEMENTS

A debugging program written by Frederick C. Druseikis, which uses the existing SNOBOL4 facilities, provided the basis for SNODDT. The command format of SNODDT was inspired by the DECSystem-10 version of DDT.

REFERENCES

1. D. R. Hanson, 'Variable associations in SNOBOL4', *Software—Practice and Experience*, **6**, 245–254 (1976).
2. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, 2nd ed., Prentice-Hall Inc., Englewood Cliffs, N.J., 1971.
3. R. B. K. Dewar, 'SPITBOL version 2.0', *SNOBOL4 Project Document S4D23*, Illinois Institute of Technology, Chicago (1971).
4. J. F. Gimpel, 'A design for SNOBOL4 for the PDP-10', *SNOBOL4 Project Document S4D29b*, Bell Laboratories, Homdel, N.J. (1973).
5. J. F. Gimpel, 'SITBOL version 3.0', *SNOBOL4 Project Document S4D30b*, Bell Laboratories, Holmdel, N.J. (1973).
6. J. F. Gimpel, 'Some highlights of the SITBOL language extensions to SNOBOL4', *SIGPLAN Notices*, **9**, 11–20 (1974).
7. R. Dunn, 'SNOBOL4 as a language for bootstrapping a compiler', *SIGPLAN Notices*, **8**, 28–32 (1973).
8. Digital Equipment Corporation, 'Fortran-10 language manual', *Publication DEC–10–LFORA–B–D*, 2nd ed., Maynard, Mass. (1974).
9. S. Arnberg *et al.*, 'DECsystem-10 Simula language handbook Part II', *Report C8399–M3(E5)*, Swedish National Defense Research Institute, Stockholm (1974).
10. Digital Equipment Corporation, 'DDT—dynamic debugging technique', *Publication DEC–10–UDDTA–A–D*, Maynard, Mass. (1975).
11. Digital Equipment Corporation, 'System reference manual', *Publication DEC–10–HGAE–D*, 3rd ed., Maynard, Mass. (1974).
12. R. E. Griswold, 'A portable diagnostic facility for SNOBOL4', *Software—Practice and Experience*, **6**, 93–104 (1975).
13. R. E. Griswold and D. R. Hanson, 'An overview of SL5', *SIGPLAN Notices* to appear (1978).
14. D. R. Hanson, 'Filters in SL5', *Comput. J.* **12**, 40–50 (1977).