A NEW APPROACH TO PROGRAM TESTING

James C. King

Computer Science Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598
USA

Keywords and Phrases

Symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation, program correctness

Abstract

The current approach for testing a program is, in principle, quite primitive. Some small sample of the data that a program is expected to handle is presented to the program. If the program produces correct results for the sample, it is assumed to be correct. Much current work focuses on the question of how to choose this sample. We propose that a program can be more effectively tested by executing it "symbolically." Instead of supplying specific constants as input values to a program being tested, one supplies symbols. The normal computational definitions for the basic operations performed by a program can be expanded to accept symbolic inputs and produce symbolic formulae as output.

If the flow of control in the program is completely independent of its input parameters, then all output values can be symbolically computed as formulae over the symbolic inputs and examined for correctness. When the control flow of the program is input dependent, a case analysis can be performed producing output formulae for each class of inputs determined by the control flow dependencies. Using these ideas, we have designed and implemented an interactive debugging/testing system called EFFIGY.

Introduction

As tools for realizing correct programs, program testing and program proving are at the ends of a spectrum whose range is the number of times the program must be executed. To establish its correctness through testing, one must execute the program at least once for all possible unique inputs; usually an infinite numb:r of times. To establish its correctness through a rigorous correctness proof, one need not execute the program at all; but he may be faced with a tedious, if not difficult, formal analysis. These two extreme points of the spectrum offer other contrasts as well. Correctness proofs usually ignore certain realities encountered in actual test runs, for example, machine dependent details like overflow and precision. (One notable effort to bring machine dependent issues into correctness proofs is the recent thesis by Sites [6]). On the other hand one may finish a proof of correctness, but seldom does he ever finish testing a program. Normal testing and correctness proofs also differ in the degree to which the user is *required* to supply a formal specification of "correct" program behavior. While a careful statement of correctness may be recommended for program testing, it is not required. A user may choose an interesting input case and then decide a posteriori, in this specific case, if the output appears to be correct. In a formal proof of correctness one must have a careful program specification.

A testing tool is described in this paper which allows one to choose intermediate points on the spectrum between individual test runs and general correctness proofs. One can perform a single "symbolic execution" of the program that is equivalent to a large (usually infinite) number of normal test runs. Test run results can be checked by careful manual inspection and if a machine interpretable program specification is supplied with the program it can be used to automatically check the results. Furthermore, by varying the degree to which symbolic information is introduced into the symbolic execution one can move from normal execution (no symbolic data) to a symbolic execution which, in some cases, provides a proof of correctness.

Symbolic Execution

The notion of symbolically executing a program follows quite naturally from normal program execution. First assume that there is a given programming language and a normal definition of program execution for that language. This execution definition must be used for production executions but an alternative symbolic execution semantics for the language can be defined to great advantage for debugging and testing. The individual programs themselves are not to be altered for testing. The definition of the symbolic execution must be such that trivial cases involving no symbols are equivalent to normal executions and information learned in a symbolic execution applies to the corresponding normal executions as well.

An execution of a procedure becomes symbolic by introducing symbols as input values in place of real data objects (e.g., in place of integers and floating point numbers). Here "inputs" is to be taken generally meaning *any* data external to the procedure, including that obtained through parameters, global variables, explicit READ statements, etc. Choosing symbols to represent procedure inputs should not be confused with the similar notion of using symbolic program variable names. A program variable may have many different specific values associated with it during a particular execution whereas a symbolic input symbol is used in the static mathematical sense to represent some unknown yet *fixed* value. Values of program variables may be symbols representing the non-specific procedure inputs.

Once a procedure has been invoked with symbolic inputs, execution can proceed as in a normal execution except when the symbolic inputs are encountered. This occurs in two basic ways: computation of an expression involving procedure inputs, and conditional branching dependent on procedure inputs.

Computation of Expressions. The programming language has a set of basic computational operators such as addition (+), multiplication (*), etc. which are defined over data objects such as integers. Each operator must be extended to deal with symbolic data. For arithmetic data this can be done by making use of the usual relationship between arithmetic and algebra. The arithmetic computations specified by these operators can be "delayed" or generalized by the appropriate algebraic formula manipulations. For example, suppose the symbolic inputs α and β are supplied as argument values to a procedure with formal parameter variables A and B. Denote the value of a program variable X by v(X). Then initially, $v(A) = \alpha$ and $v(B) = \beta$. If the assignment statement C := A + 2*B were symbolically executed in this context C would be assigned the symbolic formula $(\alpha + 2*\beta)$. The statement D := C − A, if executed next, would result in $v(D) = 2*\beta$.

Similar symbolic generalization can be done, at least in theory, for all computational operations in the programming language. In the most difficult case, one could at least *record* in some compact notation the sequence of computations which would have taken place had the arguments been non-symbolic. The success in doing this in practice depends upon how easily these recordings can be read and understood and how easily they can be subsequently manipulated and analyzed mechanically.

Conditional Branching. Consider the typical conditional program statement, the *IF* statement, taking the form:

$$\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2,$$

where B is some Boolean valued expression in the language and S_1 and S_2 are other statements. Normally, either $v(B) = true$ and statement S_1 is executed or $v(B) = false$ and statement S_2 is executed. However, during a symbolic execution $v(B)$ could be *true*, *false* or some symbolic formula over the input symbols. Consider the latter case. The predicates $v(B)$ and $\neg v(B)$ represent complementary constraints on the input symbols that determine alternative control flow paths through the procedure. For now, this case is called an "unresolved" execution of a conditional statement. The notion will be refined as the presentation develops. Since both alternatives paths are possible the only complete approach is to explore both: the execution forks into two "parallel" executions, one assuming $v(B)$, the other assuming $\neg v(B)$.

Assume the execution has forked at an unresolved conditional statement and consider the further execution for the case where $v(B)$. The execution may arrive at another unresolved conditional statement execution with associated Boolean, say C. Expressions $v(B)$ and $v(C)$ are both over the procedure input symbols and it is possible that either $v(B) \supset v(C)$ or $v(B) \supset \neg v(C)$. Either implication being *true* would show that the assumption made at the first unresolved execution, namely $v(B)$, is strong enough to resolve the subsequent test, namely to show that either $v(C)$ or $\neg v(C)$.

Because the assumptions made in the case analysis of one unresolved conditional statement execution may be effective in resolving subsequent unresolved statement executions they are preserved as part of the execution state, along with the variable values and the statement counter, and are called the "path condition" (denoted **pc**). At the beginning of a program execution the **pc** is set to *true*. The revised rule for symbolically executing a condition statement with associated Boolean expression B is to first form $v(B)$ as before and then form the expressions:

$$\mathbf{pc} \supset v(B)$$
$$\mathbf{pc} \supset \neg v(B).$$

If **pc** is not identically *false* then at most one of the above expressions is *true*. If the first is *true* the assumptions already made about the procedure inputs are sufficient to completely resolve this test and the execution follows only the $v(B)$ case. Similarly if the second expression is *true* it follows the $\neg v(B)$ case. Both of these cases are considered "resolved" or non-forking executions of the conditional statement.

The remaining case when neither expression is *true* is truly an unresolved (forking) execution of the conditional statement. Even given the earlier constraints on the procedure inputs (**pc**), $v(B)$ and $\neg v(B)$ are both satisfied by some non-symbolic procedure inputs. As discussed above, unresolved conditional statement executions fork into two parallel executions. One when $v(B)$ is assumed, in which case the **pc** is revised to $\mathbf{pc} \wedge v(B)$, the other when $\neg v(B)$ is assumed and then **pc** becomes $\mathbf{pc} \wedge \neg v(B)$. Note that the forking is a property of a conditional statement *execution* not the statement itself. One execution of a particular statement may be resolved yet a later execution of the same statement may not.

The **pc** is the accumulator of conditions on the original procedure inputs which determine a unique control path through the program. Each path, as forks are made, has its own **pc**. No **pc** is ever identically *false* since the original **pc** is *true* and the only changes are of the form $\mathbf{pc} := \mathbf{pc} \wedge q$ and those only in the case when $\mathbf{pc} \wedge q$ is satisfiable $((\mathbf{pc} \wedge q) = \neg(\mathbf{pc} \supset \neg q)$ which is satisfiable if $\mathbf{pc} \supset \neg q$ is *not* a theorem). Each path also has a unique **pc** since none are identically *false* and they all differ in some term, one containing a q the other a $\neg$q.

Symbolic Execution Tree

One can characterize the symbolic execution of a procedure by an "execution tree." Associate with each program statement *execution* a node and with each transition between statements a directed arc connecting the associated statement nodes. For each forking conditional statement execution, the associated execution node has more than one arc leaving it labeled by and corresponding to the path choices made in the statement. In the previous discussion of IF statements there were two choices, one corresponding to $v(B)$ and one corresponding to $\neg v(B)$. The node associated with the first statement of the procedure would have no incoming arcs and the terminal statement of the procedure (RETURN or END statement) is represented by a node with no outgoing arcs.

Also associate the complete current execution state, i.e., variable values, statement counter, and **pc** with each node. In particular, each terminal node will have a set of program variable values given as formulae over the procedure input symbols, and a **pc** which is a set of constraints over the input symbols characterizing the conditions under which those variable values would be computed. A user can examine these symbolic results for correctness as he would normal test output or substitute them into a formal output specification which should then simplify to *true*.

The execution tree for a program may be infinite if the program contains a loop whose number of iterations is dependent on procedure inputs. It is this fact that prevents symbolic execution from directly providing a proof of correctness technique. Symbolic execution is indeed an *execution* and at least in this simplest form described here provides an advanced *testing* methodology. Topor and Burstall ([7], [8]) have independently developed the notion of symbolic execution and added the required induction step needed to have a complete proof of correctness method. Deutsch [1], also independently, developed the notion of symbolic execution as an implementation technique for an interactive program prover based on Floyd's method [2]. In fact, one can see the basic elements of the notion of using symbolic execution as the basis for a correctness method in the earlier work of Good [3]. The author and his colleagues have been pursuing the idea of symbolic execution in its own right as a debugging/testing technique. A particular system we have built called EFFIGY is described briefly in the next section.

EFFIGY — An Interactive Symbolic Executor

The author and his colleagues at IBM Research have been developing an interactive symbolic execution system for testing and debugging programs written in a simple PL/I style programming language. The language is restricted to integer valued variables and vectors (one dimensional arrays). It has many interactive debugging features including execution tracing, break-points, and state saving and restoring. Of course, it provides symbolic execution and it uses a formula manipulation package and theorem prover developed previously by the author [4, 5].

The general facilities and capabilities available are all that is of real interest and these are perhaps simplest and most economically explained by a system demonstration. An APPENDIX is included which shows an actual script (annotated in italics) from such a demonstration. A method for exploring execution trees with their multitude of forks and parallel executions is up to the user. He is provided the ability to choose particular forks at unresolved conditional statement executions (via `go true`, `go false`, and `assume`) and also has the state save/restore ability so that he may return to unexplored alternatives later. We are currently experimenting with various "test path-managers" which would embody some heuristics for automating this process, exhaustively exploring all the "interesting" paths. As with previous testing methods the crucial issue is: if one cannot execute all cases, which ones should he do; which are the interesting ones.

We are also working on practical methods for dealing with more advanced programming language features such as pointer variables. While, as mentioned above, most such enhancements are straightforward "in theory" many offer fundamental problems in practice.

Conclusion

Interactive debugging/testing systems are powerful, useful tools for program development. A symbolic execution capability added to such a system is a major improvement. The normal facilities are always available as a special case. In addition, the basic system components of a symbolic executor provide a convenient toolbox for other forms of program analysis, including program proving, test case generation, and program optimization. Since such a system does offer a natural growth from today's systems, an evolutionary approach for achieving the systems of tommorrow is available. Valuable user experience and support is also provided. While practical use of the EFFIGY system is still quite limited, considerable insight into and understanding of the general notion of symbolic execution has been gained during its construction.

Acknowledgments

The colleagues at IBM Research collaborating with me in this work are: S. M. Chase, A. C. Chibib, J. A. Darringer, and S. L. Hantler. They have all contributed significantly to the ideas presented here and to the design and implementation of our EFFIGY system. We also appreciate the support and encouragement received from D. P. Rozenberg, P. C. Goldberg, and P. S. Dauber. The manuscript was typed by J. M. Hanisch.

References

[1] Deutsch, L.P. An interactive program verifier, Ph.D. dissertation, Dept. Comp. Sci., Univ. of Calif., Berkeley CA., May 1973.

[2] Floyd, R.W. Assigning meanings to programs, Proc. Symp. Appl. Math., Amer. Math. Soc., vol. 19, pp. 19-32, 1967.

[3] Good, D.I. Toward a man-machine system for proving program correctness, Ph.D. dissertation, Comp. Sci. Dept., Univ. of Wisc., Madison, Wisc., June 1970.

[4] King, J.C. and Floyd, R.W. An interpretation oriented theorem prover over integers, Journal of Comp. and Sys. Sci., vol. 6, no. 4, August 1972, pp. 305-323.

[5] King, J.C. A program verifier, IFIP Congress 71 Proc., Aug. 1971, pp. 235-249.

[6] Sites, R.L. Proving that computer programs terminate cleanly, Ph.D. dissertation, Comp. Sci. Dept., Stanford Univ., Stanford, CA., May 1974.

[7] Topor, R. W. Interactive program verification using virtual programs, Ph.D. dissertation, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, February 1975.

[8] Topor, R. W. and Burstall, R. M. Verification of programs by symbolic execution - progress report, unpublished report, Dept. of Mach. Intelligence, Univ. of Edinburgh, Edinburgh, Scotland, December 1973.

Appendix

A script from an actual EFFIGY session is shown below. The user's inputs are in lowercase letters and the system responses are in uppercase letters. To prevent any possible confusion the symbol "▶" is shown here to the left of the user inputs. Explanatory comments, in italic letters, have been added.

When EFFIGY is initially invoked it is in an "immediate" mode and will execute statements as they are typed. Any statement executed in this context is considered part of a main initial procedure called MAIN. The concept of the MAIN procedure and the concept of immediate execution are distinct since statements can also be executed in an immediate mode in the context of other procedures. MAIN is unique in that it has an immediate mode only and it is the only procedure privileged to execute the managerial system commands. Programs are made available to EFFIGY for stored program execution by declaring them, in MAIN, with the PROC statement similar to the way that internal procedures are declared in PL/I. However, EFFIGY does consider all procedures as EXTERNAL and they must be declared in MAIN.

Procedures are tested by a CALL from MAIN. Symbolic inputs can be supplied by enclosing a symbol string in double quotes, e.g., "a", "Dog". These symbolic constants can be used in most places instead of integer constants. The system responses drop the quotes since the context always makes the distinctions between different uses of identifiers quite clear. *Values* always involve the input symbols and never program variable names. Formulae are stored internal to EFFIGY in a "normalized" form and some of the expressions may appear quite different from what one might expect (e.g., A < 0 will be typed out as A¬> − 1). The formulae are also kept in a simplified form (e.g., 2*B = 4 is stored as B − 2 = 0).

EFFIGY runs on CMS under VM/370 on an IBM/370 model 168. The CMS filing system and context editor are used as an integral part of EFFIGY for creating, changing, and storing procedures and command files. The INPUT command directs EFFIGY to read its input from the designated file (files have two part names in CMS) instead of directly from the user's terminal. As procedures are entered into EFFIGY (by a PROC ... END declaration) the statements are sequentially numbered. These statement numbers are used to reference particular points in the procedure for inserting breakpoints, turning tracing on and off, etc.

```
▶effigy                 Invoke the EFFIGY system.
 EFFIGY  READY
▶edit absolute effigy;
                        Invoke the CMS file editor and
                         type-in a new file called
                         "absolute effigy".
 NEW FILE:
▶input
▶absolute: proc(x,y);
▶    dcl (x,y) integer;
▶    if  x<0  then  y = -x
▶             else  y =  x;
▶  end;
▶                       (end of input signified by null line.)
▶file                   Save file permanently &
                         go back to EFFIGY.
▶input absolute effigy;
                        Have EFFIGY read input from
                         that file.
    1: ABSOLUTE: PROC(X,Y);
    2:    DCL (X,Y) INTEGER;
                        Statements are numbered by EFFIGY.
    3:      IF  X<0  THEN  Y = -X
    4:               ELSE  Y =  X;
    5:    END;
                        Last line of file -- back to
                         terminal input.
▶dcl z integer;
                        Declare a variable in MAIN.
▶call absolute(55,z); display z;
                        Try a numeric execution.
 55                     Result of display statement.
▶call absolute(-66,z); display z;
 66
▶in absolute; turn all on all;
                        All tracing on in proc. "absolute".
▶in main;               Set back to MAIN.
▶call absolute("a",z);
                        Try a symbolic input "a".
   1: ABSOLUTE: PROC(X,Y);
                        Each statement execution is traced
                         by printing it.
   2: DCL (X,Y) INTEGER;
   3: IF X < 0 THEN Y = - X
 ((A¬>-1))              Evaluated result of X<0--v(B).
                         (A≤-1 or A<0)
 TYPE GO TRUE OR GO FALSE
                        Unresolved (forking) IF--User option.
▶before 5;              Stop before executing statement 5.
▶save;                  Save the current execution state.
```

```
  STATE    1 SAVED
                        EFFIGY calls this state 1.
▶go true;               Follow case where A<0.
 ((A¬>-1))              v(X<0)--evaluated test.
 TRUE BRANCH
 Y=-A                   Result of assignment to Y.
 STOPPED BETWEEN    3 AND    5
                        Stopped "before 5".
▶display variables, pc;
                        All local values and the pc.
 IN ABSOLUTE
 X=A
 Y=-A
 ((A¬>-1))              Current pc.
▶restore 1;             Return to execution state 1,
                         and try else path.
 STATE    1 RESTORED. IN ABSOLUTE
▶go false;
 ((A¬>-1))              v(X<0).
 FALSE BRANCH
   4: ELSE Y = X;
 Y=A                    New value of Y.
 STOPPED BETWEEN    4 AND    5
                        Before 5.
▶display variables, pc;
 IN ABSOLUTE
 X=A
 Y=A
 ((A¬<0))
▶xgo;                   Resume execution and delete breakpoint.
  BACK FROM ABSOLUTE TO MAIN
▶display z;
 A
▶in absolute; turn all off all;
                        Turn all tracing off.
▶in main;
▶erase pc;              Reset pc to true.
▶call absolute("a" - "b",z);go true;
 TYPE GO TRUE OR GO FALSE
                        Go true above anticipates question.
▶display z;
 -A+B
▶edit absolute effigy;
                        Invoke editor to change absolute.
▶next                   Edit command to look at line 1 of file.
▶change /absolute/newabs/
                        Change proc name.
▶bottom                 Go to end of file.
▶up 1                   Well not quite.
▶input   assert(y eq abs(x));
                        Insert a correctness specification.
▶file newabs            File away as newabs effigy.
                        Go back to EFFIGY.
▶input newabs effigy;
                        Enter into EFFIGY.
    1: NEWABS: PROC(X,Y);
    2:     DCL  (X,Y) INTEGER;
    3:     IF  X<0  THEN  Y = -X
    4:              ELSE  Y =  X;
    5:   ASSERT(Y EQ ABS(X));
                        New statement.
    6:   END;
▶erase pc;
▶call newabs("a",z); go true;
 TYPE GO TRUE OR GO FALSE
                        Response was anticipated on
                         previous line.
 ((abs(A)+A =0)) :: TRUE
                        Result of executing assert (statement 5).
                         Of form l :: r where...
                         l is evaluated assertion and
                         r is result of pc ⊃ l.
▶display z, pc;
 -A                     Value of z
 ((A¬>-1))              and pc.
▶erase pc;
▶call newabs("a",z); go false;
```

```
 TYPE GO TRUE OR GO FALSE
                        Try only other case.
 ((abs(A)-A =0)) :: TRUE
                        That also gets proved.
                        Have correctness proof--both paths
                         correct.
 A
 ((A¬<0))
▶erase pc;
▶input times effigy;
                        Now read in procedure times.
   1: TIMES:PROC(X,Y,Z);
   2:    DCL (X,Y,Z) INTEGER;
   3:    Z=0;
   4:    IF X<0 THEN
   4:    DO;
   5:       CALL ABSOLUTE(X,X);
                        Times calls absolute.
   6:       Y=-Y;
   7:    END;
   8: L:
   8:    IF X>0 THEN
                        It multiplies by looping add.
   8:    DO;
   9:       X=X-1;
  10:       Z=Z+Y;
  11:       GO TO L;
  12:    END;
  13: END;
▶call times(3,5,z); display z;
                        Try some numbers.
 15
 call times(-3,5,z); display z;
 -15
 call times(-34,"b",z); display z;
                        A mixed case--determinate control flow.
 -34*B
▶in times; turn all on 4 5 6 8 9 10;
▶before 13; in main;
▶call times("a","b",z);
                        The completely symbolic case.
   4: IF X < 0 THEN DO;
 ((A¬>-1))
 TYPE GO TRUE OR GO FALSE
▶save;
  STATE   2 SAVED
▶go true;
 ((A¬>-1))
 TRUE BRANCH
   5: CALL ABSOLUTE(X,X);
                        Executed a resolved IF in absolute.
                         Knows A≤-1.
   6: Y = - Y;
 Y=-B
   8: L: IF X > 0 THEN DO;
 ((A¬>-1))
 TRUE BRANCH
                        Another resolved IF.
                        A≤-1 so -A>0.
   9: X = X - 1;
 X=-A-1
  10: Z = Z + Y;
 Z=-B
   8: L: IF X > 0 THEN DO;
 ((A¬>-2))
 TYPE GO TRUE OR GO FALSE
▶go true;
                        Loop around.
 ((A¬>-2))
 TRUE BRANCH
   9: X = X - 1;
 X=-A-2
  10: Z = Z + Y;
 Z=-2*B
   8: L: IF X > 0 THEN DO;
 ((A¬>-3))
 TYPE GO TRUE OR GO FALSE
```

```
▶go false;        Now go out to end of proc.
 ((A¬>-3))
 FALSE BRANCH
 STOPPED BETWEEN   8 AND   13
                  Breakpoint at end of proc.
▶display variables, pc;
 IN TIMES
 X=-A-2
 Y=-B
 Z=-2*B
 ((A =-2))        Path choices determine A=-2.
▶restore 2;
 STATE   2 RESTORED. IN TIMES
▶go false;        Try another case.
 ((A¬>-1))
 FALSE BRANCH
   8: L: IF X > 0 THEN DO;
 ((A¬<1))
 TYPE GO TRUE OR GO FALSE
▶assume("a">4);
                  Add this assumption to the pc.
▶go;              Now retry the IF with new pc.
 ((A¬<1))
 TRUE BRANCH      New pc resolves it.
   9: X = X - 1;
 X=A-1
  10: Z = Z + Y;
 Z=B
   8: L: IF X > 0 THEN DO;
 ((A¬<2))
 TRUE BRANCH      Assume carries us through this one too.
   9: X = X - 1;
 X=A-2
  10: Z = Z + Y;
 Z=2*B
   8: L: IF X > 0 THEN DO;
 ((A¬<3))
 TRUE BRANCH
   9: X = X - 1;
 X=A-3
  10: Z = Z + Y;
 Z=3*B
   8: L: IF X > 0 THEN DO;
 ((A¬<4))
 TRUE BRANCH
   9: X = X - 1;
 X=A-4
  10: Z = Z + Y;
 Z=4*B
   8: L: IF X > 0 THEN DO;
 ((A¬<5))
 TRUE BRANCH
   9: X = X - 1;
 X=A-5
  10: Z = Z + Y;
 Z=5*B
   8: L: IF X > 0 THEN DO;
 ((A¬<6))
 TYPE GO TRUE OR GO FALSE
                  Unresolved when X gets to A-5.
▶go false;        Leave loop
 ((A¬<10))
 FALSE BRANCH
 STOPPED BETWEEN   8 AND   13
▶display variables, pc;
 IN TIMES
 X=A-5
 Y=B
 Z=5*B
 ((A =5))
▶restore 2;       Go back and try another case.
 STATE   2 RESTORED. IN TIMES
▶assume ( "a" eq "b" & "b" eq 2);
                  Indirectly assume A is 2.
▶go;              Does that assume resolve the if?
 ((A¬>-1))
 FALSE BRANCH     Yes it does.
```

```
   8: L: IF X > 0 THEN DO;
 ((A¬<1))
 TRUE BRANCH      This one resolved too.
   9: X = X - 1;
 X=A-1
  10: Z = Z + Y;
 Z=B
   8: L: IF X > 0 THEN DO;
 ((A¬<2))
 TRUE BRANCH
   9: X = X - 1;
 X=A-2
  10: Z = Z + Y;
 Z=2*B
   8: L: IF X > 0 THEN DO;
 ((A¬<3))
 FALSE BRANCH
 STOPPED BETWEEN   8 AND   13
▶display variables, pc;
 IN TIMES
 X=A-2
 Y=B
 Z=2*B            Result still in symbolic terms.
 ((A-B =0&B =2))
▶assert(z eq 4);
                  Does it know Z is really 4.
 ((B =2)) :: TRUE
                  Yes.
▶go;              Go on out of times.
▶
 IN MAIN          Response to previous null line.
▶edit times effigy;
                  Edit times procedure.
▶next             In editor.
▶input    assume(x eq "x0" & y eq "y0");
                  Insert correctness specifications.
▶bottom
▶up 1
▶input assert(z eq "x0" * "y0");
▶file             Replace original procedure
                    and go back to EFFIGY.
▶erase times;
                  Can't have two times routines.
▶erase pc;
▶input times effigy;
                  Input from times file.
    1: TIMES:PROC(X,Y,Z);
    2:    ASSUME(X EQ "X0" & Y EQ "Y0");
                  Used to name input values.
    3:    DCL (X,Y,Z) INTEGER;
    4:    Z=0;
    5:    IF X<0 THEN
    5:    DO;
    6:        CALL ABSOLUTE(X,X);
    7:        Y=-Y;
    8:    END;
    9: L:
    9:    IF X>0 THEN
    9:    DO;
   10:        X=X-1;
   11:        Z=Z+Y;
   12:        GO TO L;
   13:    END;
   14: ASSERT(Z EQ "X0" * "Y0");
                  Relate input values to output.
   15: END;
▶in times; turn all on 5 9 14;
                  Selectively trace.
▶in main;
▶assume("a">4 & "a"<5);
                  No integer between 4 and 5.
 CONTRADICTING ASSUMPTION. IGNORED.
▶assume("a">4 & "a"<7);
                  How about A is 5 or 6.
▶call times("a","b",z);
   5: IF X < 0 THEN DO;
 ((A¬>-1))
 FALSE BRANCH
                For 5 and 6 X > 0.
   9: L: IF X > 0 THEN DO;
 ((A¬<1))
 TRUE BRANCH
                For 5 and 6 loop some too.
   9: L: IF X > 0 THEN DO;
 ((A¬<2))
 TRUE BRANCH
   9: L: IF X > 0 THEN DO;
 ((A¬<3))
 TRUE BRANCH
   9: L: IF X > 0 THEN DO;
 ((A¬<4))
 TRUE BRANCH
   9: L: IF X > 0 THEN DO;
 ((A¬<5))
 TRUE BRANCH
   9: L: IF X > 0 THEN DO;
 ((A¬<6))
 TYPE GO TRUE OR GO FALSE
                Now must decide 5 or 6.
▶go true;       Pick 6.
 ((A¬<6))
 TRUE BRANCH
   9: L: IF X > 0 THEN DO;
 ((A¬<7))
 FALSE BRANCH   Known not > 6.
  14: ASSERT(Z EQ X0 * Y0);
 ((6*B-X0*Y0 =0)) :: TRUE
                Results check by assert--O.K.
▶display pc;
                What is the pc?
  ((A =6&A-X0 =0&B-Y0 =0))
                Relates the symbolic inputs to the
                 names given to inputs by assume
                 in the procedure.
▶display variables;
                MAIN has variables and values too.
 IN MAIN
 ABSOLUTE=PROC
                Value "PROC" means it is a procedure.
 Z=6*B
 NEWABS=PROC
 TIMES=PROC
▶quit           Leave EFFIGY system.
```