THE CONCEPT OF SELF-CHECKING PROGRAMS

Behrooz Parhami
Arya-Mehr University of Technology
Tehran, Iran

It is a known fact that digital logic circuits can be designed to be *self-checking* or *fault-secure* with respect to a class of hardware failures; i.e., such that internal failures either do not affect the circuit's behavior or are detectable by external checkers. The same idea can be applied to *redundant programs* through definition of a suitable class of *program faults* and derivation of a set of synthesis rules.

In this note, we consider a technique for making FORTRAN programs self-checking with respect to hardware or software faults affecting only a single source program statement. These include insertion, deletion, erroneous representation, or misinterpretation of a single program statement. Our discussion will be limited to statement-level redundancy although in many cases the self-checking property may also be provided through functional checks (e.g., checking of matrix inversion through subsequent matrix multiplication).

A subset of FORTRAN in terms of statement types is considered here. The allowed statement types and their self-checking replacements are given in the following table. In studying the table, these points must be taken into account:

(1) The original variable names are shorter than the system's limit and do not end with a numeral. This simplifies forming unique names for redundant check variables by adding a numeral to the end of an existing name.

(2) Statement numbers are also short and do not end in 9 so that adding a 9 to the end of each results in a new unique statement number.

(3) The statement number ε denotes the start of an error-handling routine. This routine need not be redundant as it is used only when it is known to be fault-free; i.e., when some other fault is present.

(4) PROGRAM and END statements are left unchanged since an error in these statements is either corrected by the compiler or results in appropriate diagnostic message.

(5) All forms of control transfer are checked by setting up a check variable before the transfer statement and subsequently comparing its value with a check constant at the jump target.

(6) Errors in all forms of conditional transfers are avoided by checking the conditions twice before the actual transfer can take place.

(7) Several types of checking are required for DO loops. The value of loop index is checked immediately after entering the loop and in the start of each subsequent iteration. The terminating statement number is changed so that the replaced loop does not end in a transfer statement (also see CONTINUE).

(8) It is assumed that each loop ends with its own CONTINUE statement. Proper termination of the loop is checked by comparing the final value of loop index with the loop parameters stored in auxiliary variables.

Many other statement types such as *arithmetic* IF and *computed* GO TO can be easily dealt with. But FUNCTION, SUBROUTINE, and CALL need more detailed considerations which are beyond the scope of this note. The cost of making programs self-checking is a more than threefold increase in program size and only slightly less redundancy in execution time.

Example of Statement	Self-Checking Replacement*	Note
INTEGER X,Y(8)	INTEGER X,X1,Y(8),Y1(8) INTEGER X,X1,Y(8),Y1(8)	

X=Y(4)+X**TA	X=Y(4)+X**TA
	X1=Y1(4)+X1**TA1
	IF(X .NE. X1) GO TO ε
		1

GO TO 5	L5=5 GO TO 5 GO TO 5	

5 *any statement*	L5=5
	5 IF(L5 .NE. 5) GO TO ε
	L5=0
	self-checking replacement
		2

READ 6,X,Y(4)	READ 6,X,Y(4) READ 69,X1,Y1(4)	3
6 FORMAT(10X,I4,I8)	6 FORMAT(10X,I4,I8) 69 FORMAT(10X,I4,I8)	

IF(X .GT. Y)GO TO 5	Lα=α
	IF(.NOT.(X.GT.Y))GO TO α
	L5=5
	IF(X1 .GT. Y1)GO TO 5
	α IF(Lα .NE. α)GO TO ε
	Lα=0
	IF(X1 .GT. Y1)GO TO ε
		4

DO 7 I=*m*,*n*,*k*	N7=*n*
	K7=*k*
	M7=*m*-*k*
	J7=M7
	DO 79 I=*m*,*n*,*k*
	IF(I .NE. (M7+*k*))GO TO ε
	IF(J7 .NE. M7)GO TO ε
	M7=I
	I1=I
		5

7 CONTINUE	L7=7
	7 IF(L7 .NE. 7)GO TO ε
	L7=0
	79 J7=M7
	IF((N7-M7).GE.K7)GO TO ε
	IF(N7 .LT. M7)GO TO ε
		6

Notes on the replacement table:

(1) This is similar to subsystem duplication with output checking in the case of hardware. The comparison statement may be eliminated at the risk of missing some compensating errors.
(2) We check to see if this statement is reached from a legal path (FORMAT and CONTINUE statements are excepted). One is not allowed as a statement number.
(3) Duplicate input and output data is assumed.
(4) We assume that α is a unique number not appearing as a statement label anywhere else in the program. Each such substitution results in a different value for α.
(5) Loop parameters are saved in auxiliary loop variables in order to check proper execution and termination of the loop.
(6) We assume that the CONTINUE statement is only used to terminate DO loops and each DO loop has its own terminating CONTINUE.
(*) No special compiler feature is assumed.