SCIENCE RESEARCH COUNCIL

RUTHERFORD LABORATORY
ATLAS COMPUTING DIVISION

F R 8 0   T E C H N I C A L   P A P E R   2 6

FR80 DRIVER and Provably Safe Programs

issued by
R W Witty

7 March 1977

DISTRIBUTION:	F R A Hopgood	J R Gallop
	L O Ford	D C Sutcliffe
	A H Francis	R E Thomas
	D A Duce	R W Witty
		(Dr T Anderson, Newcastle University)

1. PREFACE

This paper outlines the principles of safe programming invented by Dr Tom Anderson of Newcastle University (1) and imported by John Rushby.

FR80 SYSLOG and FR80 DRIVER are both (famous last words) examples of safe programs. FR80 DRIVER is implemented in the language DRIL which was specifically designed for safe programming (2).

2. INTRODUCTION

The provision of a proof of the correctness of a program should increase one's confidence that, when executed, the program's behaviour will conform to its specification. Such a proof should consist of:

(a) the specification of the program's intended function

(b) an argument to show that the program will always meet its specification. This argument should have two parts, firstly a proof of termination and secondly, a proof that if the program terminates then it will have behaved according to its specification.

To be convinced that the program is indeed correct one must be satisfied that both the specification and the argument are correct. Unfortunately current experience indicates that correctness proofs constructed for even quite short programs can be both lengthy and complex. For a proof to be of any real value it must be clearer and simpler than its associated program.

Consider the following specification:

Using only integer arithmetic, find the largest integer i less than or equal to the square root of a given non-negative integer n. That is, given $n \geqslant o$, find:

$$i = \lfloor \sqrt{n} \rfloor = (i^2 \leqslant n) \wedge ((i+1)^2 > n)$$

The following solution is based on the Newton Raphson method.

```
SOL1 : begin
              i : = $\lfloor \frac{n+3}{2} \rfloor$   { initial value } ;
       while  $i^2 > n$
              do begin
                   i : = $\lfloor \frac{n+i^2}{2i} \rfloor$   { next estimate } ;
              end;
       end
```

A proof of correctness consists of:

Proof of termination:

1. The while loop must be entered at least once since $\lfloor \frac{n+3}{2} \rfloor^2 > n$ for any integer n. Hence on entry to the loop $i^2 > n$ which together with $i > o$ implies that $i > \lfloor \frac{n+i^2}{2i} \rfloor$. Replacement of i by this value at each iteration ensures that the value of i must strictly decrease, and as it is always non-negative, only a bounded number of iterations can therefore occur.

Proof of correct behaviour:

1. The while loop must be entered at least once and must terminate (see above).

2. After termination, $i^2 \leqslant n$ (from the while test) and also $i = \lfloor \frac{n+j^2}{2j} \rfloor$ where j denotes the penultimate value held by i.

3. Now $\left(\lfloor \frac{n+j^2}{2j} \rfloor + 1 \right)^2 > n$ for any $j \neq o$

   therefore $(i+1)^2 > n$ and $i^2 \leqslant n$

The above proof is very informal, and some simple lemmas on integers have been omitted. Even so, it fails to inspire a great deal of confidence. If a proof is to be of any real value it must be clearer and simpler than its associated program. For just as a simple program is more likely to be correct than a complex program, so a simple proof is more likely to be valid than a complex proof.

3. ADEQUATE PROGRAMS

Dijkstra's proof guided program design methodology (3) helps to create simpler proofs. A variation of this technique is instead of attempting to prove the correctness of a program with respect to its original specification, some weaker criterion of acceptable behaviour is selected. That is, if the program's original specification is denoted by P then a specification Q is chosen such that:

(1) any program which conforms to P will also conform to Q

(2) Q prescribes an acceptable behaviour of the program. The program is then designed and constructed in an attempt to conform to P, but so as to facilitate a much simpler proof of correctness with respect to Q than would be possible using P. This will be termed a proof that the program is adequate.

4. SAFE PROGRAMS

In the context of software reliability a special case of adequacy, termed safeness, is relevant. As a weaker specification for a program intended to satisfy P, take Q to be P v 'error', meaning that the program should either behave as was originally intended or should terminate giving a reason for its failure. A proof of adequacy for this particular Q will be termed a proof that the program is safe.

Ideally a program should be designed so that its proof of safeness is substantially simpler than its correctness proof. One way of achieving this objective is shown in the following solution to the largest square root problem introduced above.

Safe specification:

find i such that $[(i^2 \leq n)) \wedge ((i+1)^2 > n)] \vee$ 'error'

Program:

```
SOL2 : begin
        i : = initial value;

        iteration-counter := 0;

        while  ((i^2 > n) or (i + 1)^2 ≤ n)) and iteration-counter
                                                < iteration-limit
            do  begin
            i := next estimate ;

            iteration-counter := iteration-counter + 1;

            end;

        safety check 1:
        if iteration-counter ≥ iteration-limit then error ('loop-limit');

        safety check 2:
        if not ((i^2 ≤ n) and ((i + 1)^2 > n)) then error ('wrong answer');

    end
```

Proof of safeness:

Termination:

Guaranteed by testing of iteration-counter

Adequacy:

After termination either an error will have been detected or a correct answer will have been calculated as an explicit test of correctness is included.

The simple nature of this proof leaves little opportunity for error which justifies a high level of confidence in the safeness of the program. Note tht a proof of correctness assumes that the program's input conforms to the specification. It says nothing about the program's behaviour upon incorrect data. However safeness is valid for any input because the only assumptions made are actually checked at run time.

5. BOUNDED REPETITION

The above program is atypical in so far as the explicit testing of a program's results is rarely feasible in practice. However, it seems perfectly feasible to eliminate the need for a proof of termination simply by programming in languages which ensure that all programs must halt, thereby greatly simplifying the overall proof.

Such languages do not provide explicit control transfer and impose constraints on all iterative and recursive facilities. As a result they cannot be used to program all of the recursive (computable) functions, and are known as sub-recursive languages. The work of Constable and Borodin (4) indicates that such languages do include all the functions actually used in computing and this seems to be borne out in practice (see below). Indeed these restrictions are an advantage of the sub-recursive languages.

5.1 Bounded Iteration

An iterative facility provided by many languages can be denoted by:

<u>repeat</u> S <u>possibly</u> <u>forever</u>

where S denotes a statement list which may or may not include conditional <u>exit</u> s. S is repeatedly executed until an <u>exit</u> is taken and the construct is terminated. The <u>while</u> loop is a typical example of this type.

Consider two special cases of the construct.

<u>repeat</u> S <u>forever</u>

S contains no <u>exit</u> s and is repeated infinitely. This special case is rarely needed, and would deserve careful consideration if it were.

repeat S exactly n times

n denotes a non-negative integer value ; S contains no exits and is executed precisely n times. This special case is frequently needed. Its termination is guaranteed.

The main criticism of the more powerful possibly forever construct is that it permits infinite repetition when in all probability the programmer did not intend this to occur. By analogy with the two special cases above an alternative version is suggested which prevents infinite repetition

repeat S upto n times

S contains one or more conditional exits and is executed at most n times, the construct being terminated earlier if an exit is taken. Compare this with the hand-coded version of SOL2. An implementation of this scheme might allow the user to chose between handling his own loop limit errors (as in SOL2) or automatically terminating the program.

A sub-recursive language only provides bounded iteration constructs.

repeat S upto n times (S contains one or more exits)

repeat S exactly n times (S contains no exits)

If potentially infinite iteration is to be included in a programming language then a separate construct should be specially provided.

5.2 Bounded Recursion

Recursive constructs may be constrained in a similar manner to the iterative constructs.

6. EXPERIENCE WITH ADEQUATE PROGRAMS

An attempt has been made to demonstrate the possibility of writing a practical piece of software so as to obtain a simple proof of adequacy, and is described by Reynolds (9). A file system was implemented with specification P: "All user commands to the file system are correctly processed". A proof of adequacy was provided for the specification Q: "All user commands to the file system are either correctly processed, or if not, the user is sent a warning message and the integrity of all previously filed data is maintained". By means of isolating those routines which actually modifed the file structures, and incorporating run time checks to verify their actions, a reasonably simple proof of Q was obtained. A large portion of the software could be ignored completely when establishing adequacy, a considerable benefit. Another encouraging feature of this experiment was that throughout the debugging phase, when the program was patently not correct, its behaviour was, however, always adequate.

7. EXPERIENCE WITH SAFE PROGRAMS

The Rutherford Laboratory has a small mini-computer (the FR80) whose only software tools are an assembler, a loader and an ODT-like window into the core. The machine has no supervisor program, no core protection and no printer. A major difficulty in programming this machine is that erroneously looping programs generally overwrite themselves making debugging extremely difficult. Therefore it was decided to construct all new software according to the principles of safe programming. Several programs have been constructed including a multi-tasking program (5). The multi-tasking program is built with multiple exit loops based on Zahn's construct (6). These have proved a success as they eliminate the need for additional tests immediately following the loop to determine which of the possibilities caused the loop to terminate, see example 2 and (2).

None of the safe programs has failed to terminate, even during development when bugs were obviously present. The need to place a bound on each loop has proved beneficial rather than restrictive. The very act of determining the limit has caught errors. It is suprising how small most of the loop bounds are in practice. The overhead in implementing bounded loops is negligible (2). Some errors caused by punching errors, pre-processor errors and (unproved) modifications were caught at run time by safe programming. These errors could not have been caught had the programs merely been proved correct. The knowledge that a program will terminate safely whatever its input has greatly increased confidence in the programs, has saved hours of debugging time and has increased the programmers' peace of mind.

Meissner (7) too has reported favourably about bounded loops and has suggested a template from which bounded loops may be constructed in standard FORTRAN, see example 1. FOREST (8) supports bounded loops, see example 3.

8. CONCLUSION

Safeness directed program design and construction really works.

```
I = initial value

DO 7 LOOP=1,LIMIT

    IF((I**2).LE.N).AND.((I+1)**2).GT.N)GOTO 8

    I = next estimate(I)

7 CONTINUE

    CALL ERROR ('LOOP LIMIT')

8 CONTINUE
```

Example 1 Meissner's safe FORTRAN loop

```
I := initial value

loop

    terminate foundanswer if (I^2≤n) and (I+1)^2 > n endt

    I := next estimate (I)

    terminate looperror if done looplimit times

repeat

    situation looperror causes error ('loop limit') ends

    situation foundanswer causes ok ends

endloop
```

Example 2 Multiple-exit loop based on Zahn's constuct

```
I = initial value

CYCLE LOOP = 1, LIMIT

    IF((I**2).LE.N).AND.((I+1)**2).GT.N)EXIT

    I = next estimate (I)

REPEAT

    CALL ERROR ('LOOP LIMIT')

ENDCY
```

Example 3 FOREST

9. REFERENCES

1.	ANDERSON T	'Probably safe programs' University of Newcastle, Computing Laboratory, Technical Report No 70 1975
2.	WITTY R W	'FR80 DRIVER Sofware Construction' FR80 Technical Paper 21
3.	DIJKSTRA E W	'Concern for correctness as a Guiding Principle for Program Composition'. INFOTECH Fourth Generation International Computer State of the Art Report p 357 1971
4.	CONSTABLE R L, BORODIN A B	'Subrecursive Programming Languages Part 1: Efficiency & Program Structure JACM 19 p 526 1972
5.	WITTY R W	'FR80 DRIVER and the ACL SDF Subset' FR80 Discussion Paper 15
6.	ZAHN C T	'A Control Statement for Natural Top Down Structured Programming' Lecture Uses in Computer Science Vol 19 Paris 1974 ed Goos, Hartmanis
7.	MEISSNER L P	'Bounded Loops' FOR-WORD Vol 3 No1 Jan 1977
8.	DUCE D A	'FOREST A Structured FORTRAN preprocessor' PRIME User Note 3
9.	REYNOLDS M S	'An approach to the problem of writing highly reliable file system software' Internal Memorandum, Newcastle University

gm