Teaching Formal Software Engineering at Loughborough

R G Stone & D J Cooke
Department of Computer Studies, LUT

Part 1 - The satisfaction of programming

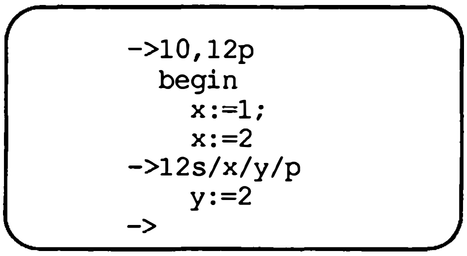
It's all new
What is the source of satisfaction in programming? One answer is to say that as a science it is new. Computer Science is exciting. Computing is still a young science. There are still many 'professionals' who were not trained in Computing. There have only been 'Chartered Engineers' in Computing for a year or so.

Progression of demands on professional
As the science is maturing the demands on the professional have changed. From 'Coding' there has been a requirement to move to 'Software Engineering', and now there is a need to move to 'Formal Software Engineering' (cf 055 standard). As each change appeared the source of satisfaction has also changed.

a) Coding/Getting my program to work!
Initially satisfaction was derived from overcoming the obstacles of actually using a computer.

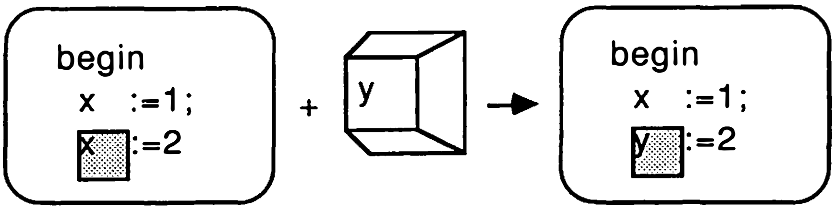
Past the editor: Editors have come a long way because getting the program 'typed in' was initially very tedious.

i) a line editor has the burden of describing the site of the intended edit and allows arbitrary (and so possibly incorrect) replacement.



```
->10,12p
  begin
    x:=1;
    x:=2
->12s/x/y/p
    y:=2
->
```

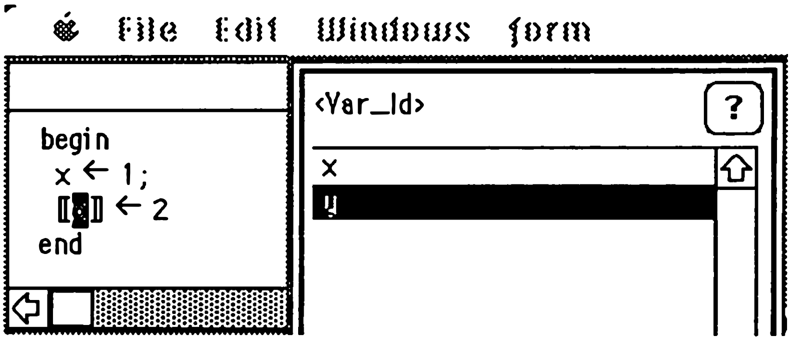
ii) a screen editor with its use of 'mouse' overcomes 'site of edit' problem but still allows arbitrary replacement.



```
begin
 x  :=1;
 x  :=2
```

```
begin
 x  :=1;
 y  :=2
```

iii) a syntax directed editor allows only syntactically correct replacement.



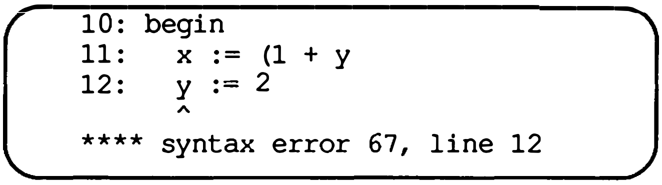
```
begin
 x ← 1;
 [[y]] ← 2
end
```

```
<Var_Id>
x
y
```

(dialog offers a scrolling list of identifiers in scope)

Past the syntax check: there is satisfaction derived from having obeyed all the strict syntactical (grammatical) rules of the programming language.

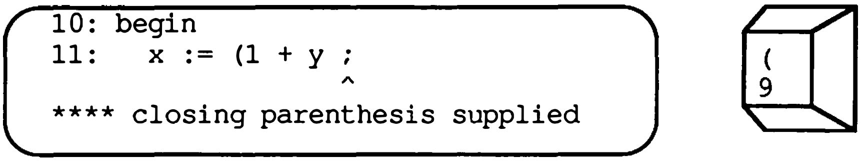
syntax error



```
10: begin
11:   x := (1 + y
12:   y := 2
      ^
**** syntax error 67, line 12
```

(but where is the error really?)

auto-correct

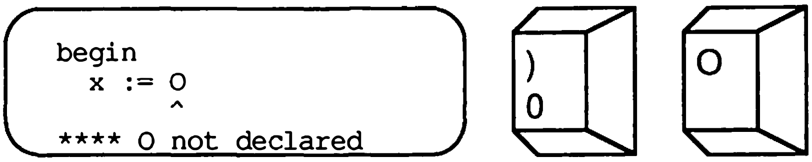


```
10: begin
11:   x := (1 + y ;
                  ^
**** closing parenthesis supplied
```

(but what did the programmer really mean?)

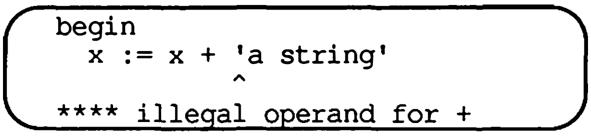
Past the static(type) check: there is satisfaction from having obeyed all the static semantic rules including those relating to 'declare before use' and 'consistent use'

... not declared



```
begin
  x := O
       ^
**** O not declared
```

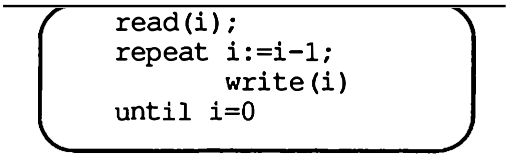
... not allowed



```
begin
  x := x + 'a string'
           ^
**** illegal operand for +
```

Past a run with simple test data: there is satisfaction derived from getting a program to run once with simple data. But what of the undiscovered 'bug'...?

Correct result when the test data was the number 5

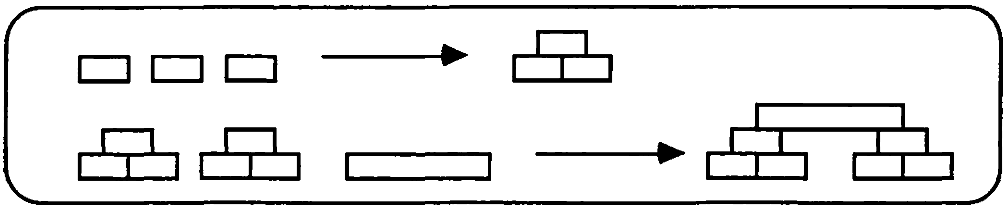


```
read(i);
repeat i:=i-1;
       write(i)
until i=0
```

b) Software Engineering/Past the Quality Assurance test

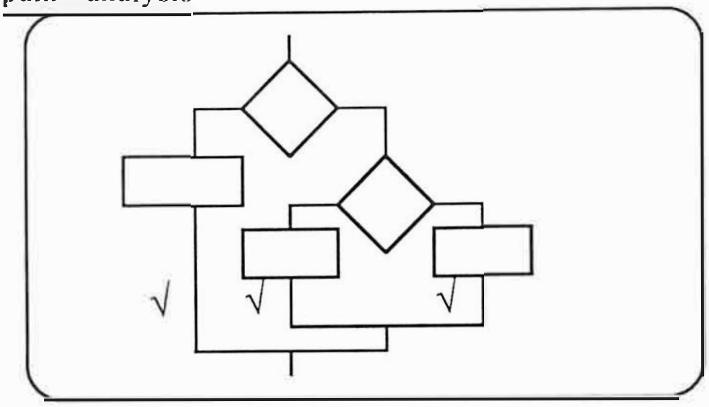
When a Q/A team is involved there is satisfaction derived from other people not being able to discover any errors in your program. Given more methodical testing there is satisfaction derived from knowing that individual modules of your program have been successful in another (smaller) environment.

module then system



There is further satisfaction to be derived from knowing that every path through your program has been checked at least once.

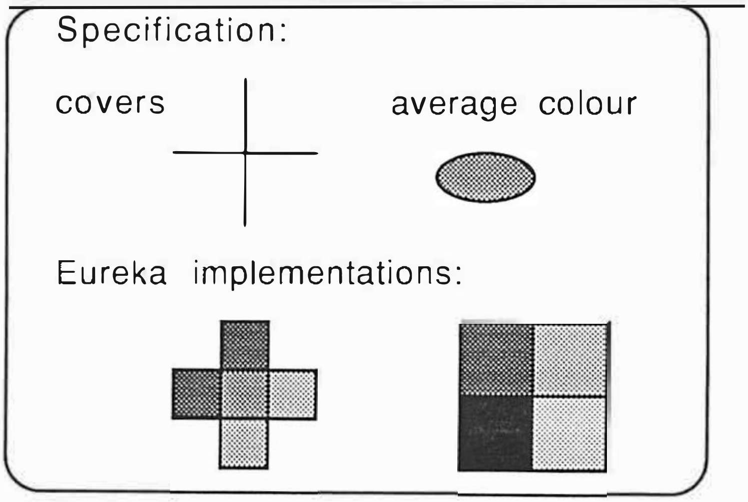
every path analysis



c) Formal Software Engineering/Proved

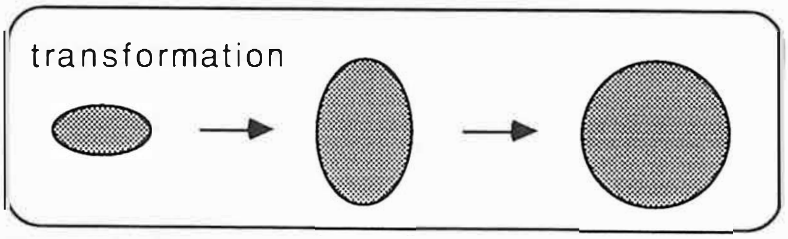
The satisfaction here derives not from the interaction with a machine (cf Coding) or with other people (Q/A) but from achieving very strict personal goals. Two different implementation paths are recognised:

invent with subsequent verification against specification



To use this path it is necessary to be clear how we are going to establish the acceptability of an implementation.

transforming specification



To use this path it is necessary to be clear what is preserved during any transformation.

Both implementation paths require formal proofs to be undertaken. The nature of formal proof is often a high level plan requiring a large number of small steps. To be attractive any machine support must do the 'housekeeping' well and leave the person free to concentrate on the high level aspects.

There is potentially a very high degree of personal satisfaction here. However the style of work (no lines of code until well into project time) and the kind of code produced (very simple, stylised) requires re-education of managers to keep them satisfied.

Part 2 - The form-tool transformer

The support we are offering may be called a 'Symbolic Calculator'. It is capable of applying a selected pattern rule to a selected (sub)expression. It keeps a record of rules applied, with restart permitted from any previous position. It allows new rules to be deduced, saved and applied like any other built-in rule.

The person using the transformer is only required to choose rule to be applied and the site where it should be applied. The tedious business of making all the relevant substitutions is handled by the machine. Any (sub)expressions that are not of interest to the current transformation can be 'hidden' using '...' but can be re-instated at any stage later as required.

For the 'Invent and verify' style the use of the transformer is:
  a) The theorem is: Implementation implies Specification
  b) Transform the expression of the theorem to 'true'

Invent and Verify example from form-tool:

Suppose that pre-f(d) is the pre-condition for function f and let post-f(d,r) specify that r is the result from f with input d. If g(h(d)) is proposed as an implementation of f(d) then it is necessary to prove that

$$\text{pre-f}(d) \Rightarrow \text{post-f}(d,\ g(h(d)))$$

For the 'Transformation' style the use of the transformer is:

a) Manipulate specification to match a standard template for which an implementation is known, or

b) Find a way of dividing the specification into smaller parts which in sequence or in parallel achieve the complete specification, and then applying the transformation style to each part.

Transformation example from form-tool:

Suppose that the specification of f is given as an equation relating d and r. The transformation consists of the algebraic process of 'making r the subject of the formula'.

(Open) Questions

a) Is there as much satisfaction in 'proving' as in the first successful 'run'?

b) Does the coding skill tend to coincide with proof skill in an individual at present?

c) Can/will the skills coincide in the future with proper training/tools

d) can machine support cover up a weakness in either coding or proving or would a coder+prover team be more appropriate?

e) Is a proof environment different from an IPSE?