Short Notes

A Control Structure for a Variable Number of Nested Loops

A recent paper by Skordalakis and Papakonstantinou proposed a new control structure with a variable number of nested loops and suggested three possible methods of implementation. Here we propose an alternative programming technique which is capable of solving this class of problem in an extremely efficient manner.

Recently Skordalakis and Papakonstantinou[1] proposed a new control structure for a variable number of nested loops. This control structure enables a large class of combinatorial type algorithms to be written in a simple non-recursive manner. Basically their structure allowed for a variable number of nested FORTRAN-type DO loops to be coded by means of arrays of loop variables, initial values, step sizes and final values. In their paper they suggested three possible implementation strategies:

(a) modifying the compiler of the host programming language to incorporate the new control structure,
(b) developing a preprocessor to translate the augmented programming language into the host programming language,
(c) implementing the new control structure through subroutine calls.

They implemented (c) with FORTRAN as the host programming language.

The purpose of this note is to propose an alternative approach which is substantially more efficient than (c) and yet does not require the complexity of (a) or (b). We propose the use of a *program generator*, coded in a language, X say, that generates a program in the host programming language containing the required number of nested control structures. In our experiments both X and the host language were Pascal, but the method is applicable to a wide range of current programming languages.

In our method the user is required to provide three procedures *precode*(*k*,*n*), *corecode*(*n*), and *postcode*(*k*, *n*). These are required to generate Pascal code for the *k*th precode etc. given a maximum depth of nesting *n*. When the program generator is executed and the user supplies the depth of nesting required, a Pascal program is automatically generated. Our implementation, given in Fig. 1, also accepts procuedures that declare and initialize any extra variables required. The nested loop structure generated has the format of Fig. 2. **While** loops are generated because of Pascal's limitations on the **for** loop step size.

To enable timing measurements to be performed, we considered the example given in Ref. 1: that of finding all combinations of q elements from a given set of p elements. The procedures supplied by the user for this problem are given in Fig. 3 and an example of a generated program in Fig. 4. The depth of nesting, q, is supplied by the user when the program generator is run. The value p is read by the generated program in its initialization phase. To prevent file transfer time dominating the measurements, the generated program was edited to sum rather than print the combinations produced.

The results of our timing measurements are summarized in Fig. 5 for values of $p = 20$ and $q = 10$. The table contains cpu times for:

(i) the program generator method considered in this note
(ii) a recursive Pascal program based on the Algol 60 program in appendix A of Ref. 1
(iii) a Pascal implementation of the subroutine method (c) based on the FORTRAN code of appendix B in Ref. 1

```
ik := bk;
while ik < = ek do begin
    {precode for loop k}
    {loop for k + 1 or corecode if k = n}
    {postcode for loop k}
    ik := ik + sk;
    end;
```

Figure 2. Basic loop structure of generated program.

```
program generator;
var
  n, k: integer;

  procedure declare (n: integer);          extern;
  procedure initialization (n: integer);   extern;
  procedure precode (k, n: integer);       extern;
  procedure corecode (n: integer);         extern;
  procedure postcode (k, n: integer);      extern;

begin
writeln (output, 'Supply the depth of nesting required');
readln (input, n);
writeln (output, 'program nest;');
writeln (output, 'var i1, b1, e1, s1');
for k := 2 to n do
   writeln (output, ',i', k:1, ',b', k:1, ',e', k:1, ',s', k:1);
writeln (output, ': integer;');
declare (n);
writeln (output, 'begin');
initialization (n);
for k := 1 to n do
  begin
  writeln (output, 'i', k:1, ':= b', k:1, ';');
  writeln (output, 'while i', k:1, ' < = e', k:1, 'do begin');
  precode (k, n);
  end;
corecode (n);
for k := n downto 1 do
  begin
  postcode (k, n);
  writeln (output, 'i', k:1, ':= i', k:1, ' + s', k:1, ';');
  writeln (output, 'end;');
  end;
writeln (output, 'end.');
end.
```

Figure 1. The Pascal program generator (this version assumes a separate compilation facility).

```
procedure precode (k, n: integer);
begin
if k < n then writeln (output, 'b', (k + 1):1, ':= i', k:1, ' + 1;');
end;

procedure corecode (n: integer);
var k: integer;
begin
write (output, 'writeln(il')';
for k := 2 to n do write (output, ', i', k:1);
writeln (output, ');');
end;

procedure postcode (k, n: integer);
begin
{empty}
end;

procedure declare (n: integer);
begin writeln (output, 'p: integer;'); end;

procedure initialization (n: integer);
var k: integer;
begin writeln (output, 'readln(p);');
writeln (output, 'b1 := 1;');
for k := 1 to n do writeln (output, 'e', k:1, ':= p; s', k:1, ':= 1;');
end;
```

Figure 3. User supplied procedures for solution of combination problem.

```
program nest;
var i1, b1, e1, s1
  , i2, b2, e2, s2
  , i3, b3, e3, s3
    : integer;
  p : integer;
begin
readln (p);
b1 := 1;
e1 := p; s1 := 1;
e2 := p; s2 := 1;
e3 := p; s3 := 1;
i1 := b1;
while i1 < = e1 do begin
  b2 := i1 + 1;
  i2 := b2;
  while i2 < = e2 do begin
    b3 := i2 + 1;
    i3 := b3;
    while i3 < = e3 do begin
      writeln (i1, i2, i3);
      i3 := i3 + s3;
      end;
    i2 := i2 + s2;
    end;
  i1 := i1 + s1;
  end;
end.
```

Figure 4. Generated program for combinations program ($q = 3$) (the code has been manually indented for clarity).

(iv) as for (iii) but with the procedure calls replaced by in-line code.

(i) Program generator method	
Time to execute program generator	0.45
Time to compile and load generated program	2.97
Time to execute generated program	3.01
	6.43
(ii) Recursive program method	
Time to execute program	35.96
(iii) Subroutine method of Ref. 1	
Time to execute program with procedure calls	92.41
(iv) Subroutine method of Ref. 1	
Time to execute program with procedure calls expanded 'in-line'	32.98

Figure 5. Timings for combinations problem with $p = 20$ and $q = 10$. Times are cpu seconds on a Prime 750, using the Pascal compiler running under Primos.

As can be seen, the program generator method proposed in this note is significantly more efficient than recursion or subroutine implementation methods. A major contribution to this efficiency results from the absence of procedure calls and array references. A further advantage over recursive implementations results from the omission of any test for the depth of recursion to decide between execution of the corecode or entering another loop. We conclude that program generation provides an efficient, flexible implementation method that is readily available in current programming languages without resort to language extensions.

B. J. McKENZIE
Department of Computer Science,
University of Canterbury,
Christchurch, New Zealand.

TADAO TAKAOKA
Department of Information Science,
Ibaraki University,
Hitachi, Japan

Reference

1. E. Skordalakis and G. Papakonstantinou, A control structure for a variable number of nested loops. *The Computer Journal* **25** (No. 1), 48–51 (1982).

Received December 1982

Taxonomic Studies on Current Approaches to Systems Analysis

Recently Wood-Harper and Fitzgerald[1] have presented a taxonomy for current approaches to systems analysis. This note examines the fundamental paradigms on which their scheme is based. Some changes are proposed and a major problem of such classification methods is identified.

Taxonomy deals with the classification of entities into groups dependent on the possession of a set of characteristics distinctive of the group. A mark of a good taxonomic scheme is, that on being presented with an entity we can, from its characters, place it unambiguously into a recognized taxon. In the more usual context of classification of plants and animals deciding what constitute distinguishable taxa may be difficult.[2]

Wood-Harper and Fitzgerald[1] introduced a taxonomic scheme for systems analysis methodologies expressed in terms of two paradigms, which they describe as a systems paradigm and a science paradigm, as described by Checkland.[3] The use of this latter term implies a commonality of approach of the paradigm with that of natural science. Checkland's definition is open to severe criticism on technical philosophical grounds as a characterization of science. There is no consensus among philosophers of science about what constitutes the 'science paradigm'. Checkland's definition is not one which would gain

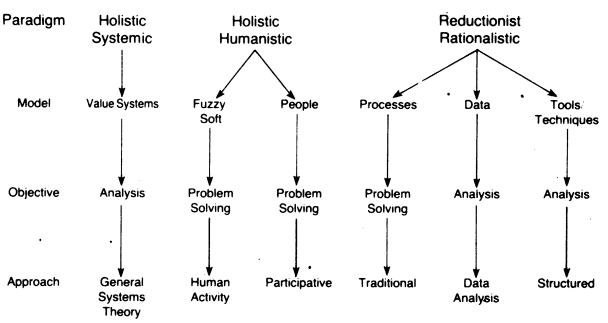

Figure 1. Revised taxonomy of systems analysis approaches (after Ref. 1).

universal assent; rather, use of the term 'science paradigm' introduces an additional highly contentious element into the discussion. This is particularly unfortunate, since there is no reason to invoke the term science in this context. A description of this paradigm which would present none of the problems of the current one would be 'rationalistic/reductionistic'. For this description Checkland's definition would be acceptable.

The second paradigm identified by Wood-Harper and Fitzgerald is also problematical. As they state, the systems paradigm is difficult to characterize, which excludes it as a basis for a taxonomic classification. One of the fundamental necessities for an acceptable taxonomy is that taxa can be unambiguously defined. An obvious alternative can be found. What characterizes general systems theory, human activity systems and participative designs is their holistic approach to system study, that is to say that they deal with problems in terms of 'wholes' rather than using an 'atomistic/reductionistic' approach. The various methods differ in the aspect of system study on which they lay particular emphasis. Two holistic approaches can be discerned; humanistic and systemic. All the methods discussed are systemic in a general sense, but general systems theory gives primacy to an abstract theory of systems, hence it may be characterized as systemic. The humanistic approach is a primary characteristic of participative design and human activity systems.

A revised taxonomy is presented in Fig. 1.

A particular problem with taxonomic schemes is that frequently they fail to take into account significant characteristics. Figure 2