```
begin
   return z.num >= i_zero;
end ">=";


function "<=" (x, y : Universal_real) return boolean is
  z : Universal_real := x - y;
begin
   return z.num <= i_zero;
end "<=";


function "<"  (x, y : Universal_real) return boolean is
  z : Universal_real := x - y;
begin
   return z.num < i_zero;
end "<";


function ">"  (x, y : Universal_real) return boolean is
  z : Universal_real := x - y;
begin
  return z.num > i_zero;
end ">";


function eql  (x, y : Universal_real) return boolean is
  z : Universal_real := x - y;
begin
  return eql(z.num, i_zero);
  end eql;

end UNIVERSAL_REAL_ARITHMETIC;
```


realized in Ada. The use of a real programming language seems to be more realistic than the use of an enriched Pascal as in [Ten 82]. Tennent uses for the two examples in his paper two different extensions of Pascal.

Section 2 introduces three principles of program structuring. In sections 3 and 4 the elements of Ada for program structuring are explained and Ada versions for quicksort and the banker's algorithm are given. These programs are compared with those of Hanson and Tennent. Conclusions are offered in section 5.

2 PRINCIPLES OF PROGRAM STRUCTURING

Before introducing the three principles of program structuring (for monolithic programs with the usual global visibility rule) we recall some properties of block structure for such programs.

Block structure generally allows the (textual) nesting of blocks.

There is one outermost block which contains all other blocks of the program. This outermost block is said to be on level 0. A block B immediately contained in a block A, which is on level n, is said to be on level n+1. We also say that B is on higher level than A. Thus the outermost block is on the lowest level.

Two blocks immediately contained in some declaration part are said to be parallel. They are on the same level.

There are two kinds of dependency between blocks:

d1) block B uses an entity (type, variable, etc) immediately declared in block A. A consequence of this is that B must be contained in A but not necessarily immediately.

d2) block A is called by block B, ie a call of A is contained in the statement part of B. A consequence of this is that A must be placed in such a manner that it is visible in the statement part of B; eg A cannot be contained in a block C contained in B.

In the following we define three principles of program structure for monolithic programs :

s1) the kind of nesting (flat/deep)

s2) the kind of ordering of parallel subprograms (alphabetical/logical)

s3) the use of subunits (yes/no)

MORE ON BLOCK STRUCTURE : USING ADA

J.F.H. Winkler

Siemens AG, Central Technology Division
D-8000 Munich 83, Fed.Rep.Germany

SUMMARY

This paper dicusses the arrangement of blocks in programs and continues a discussion initiated in two recent papers by Hanson and Tennent. In addition to the nesting of blocks which was treated by Hanson and Tennent we discuss here two more principles for the arrangement of blocks: the ordering of (the blocks of) parallel subprograms and the use of subunits. The three principles together yield a conceptual framework in which program structures can be characterized in a qualitative manner. The programming language Ada is used to formulate different variants of the examples already used by Hanson and Tennent. It is shown that Ada allows the formulation of the program structures recommended by Hanson as well as those recommended by Tennent.

1 INTRODUCTION

In two recent papers [Han 81; Ten 82] Hanson and Tennent discuss the pro's and con's of block structure in programming languages. To show the advantages and disadvantages of block structure they use two program examples : the quicksort program [Wir 76] and the banker's algorithm [Bri 73]. Hanson and Tennent deal with two aspects of programs: arrangement of blocks (of subprograms) and visibility of declared entities. With respect to the arrangement of the blocks in a monolithic program they distinguish between nested and flat structures. By a monolithic program we mean a program which contains one outermost block in which all other blocks are at least logically included. The inner blocks must not necessarily be included textually if a mechanism similar to the subunit mechanism of Ada [Ref 83] is used. This is discussed in more detail in section 2 (cf figure 6 and 9).

In this paper we concentrate on the structural aspects of such programs and discuss two further principles for the arrangement of blocks inside a program: the ordering of parallel subprograms and the use of subunits. The three structuring principles "nesting", "ordering of parallel subprograms", and "use of subunits" yield a conceptual framework within which the structure of programs can be characterized in a qualitative manner. Furthermore we show that the program structures which can be characterized by the three structuring principles can be

```
    begin
      return z.num >= i_zero;
    end ">=";

    function "<=" (x, y : Universal_real) return boolean is
      z : Universal_real := x - y;
    begin
      return z.num <= i_zero;
    end "<=";

    function "<"  (x, y : Universal_real) return boolean is
      z : Universal_real := x - y;
    begin
      return z.num < i_zero;
    end "<";

    function ">"  (x, y : Universal_real) return boolean is
      z : Universal_real := x - y;
    begin
      return z.num > i_zero;
    end ">";

    function eql  (x, y : Universal_real) return boolean is
      z : Universal_real := x - y;
    begin
      return eql(z.num, i_zero);
    end eql;

  end UNIVERSAL_REAL_ARITHMETIC;
```

MORE ON BLOCK STRUCTURE : USING ADA

J.F.H. Winkler

Siemens AG, Central Technology Division
D-8000 Munich 83, Fed.Rep.Germany

SUMMARY

This paper dicusses the arrangement of blocks in programs and continues a discussion initiated in two recent papers by Hanson and Tennent. In addition to the nesting of blocks which was treated by Hanson and Tennent we discuss here two more principles for the arrangement of blocks: the ordering of (the blocks of) parallel subprograms and the use of subunits. The three principles together yield a conceptual framework in which program structures can be characterized in a qualitative manner. The programming language Ada is used to formulate different variants of the examples already used by Hanson and Tennent. It is shown that Ada allows the formulation of the program structures recommended by Hanson as well as those recommended by Tennent.

87-6.48 III

1 INTRODUCTION

In two recent papers [Han 81; Ten 82] Hanson and Tennent discuss the pro's and con's of block structure in programming languages. To show the advantages and disadvantages of block structure they use two program examples : the quicksort program [Wir 76] and the banker's algorithm [Bri 73]. Hanson and Tennent deal with two aspects of programs: arrangement of blocks (of subprograms) and visibility of declared entities. With respect to the arrangement of the blocks in a monolithic program they distinguish between nested and flat structures. By a monolithic program we mean a program which contains one outermost block in which all other blocks are at least logically included. The inner blocks must not necessarily be included textually if a mechanism similar to the subunit mechanism of Ada [Ref 83] is used. This is discussed in more detail in section 2 (cf figure 6 and 9).

In this paper we concentrate on the structural aspects of such programs and discuss two further principles for the arrangement of blocks inside a program: the ordering of parallel subprograms and the use of subunits. The three structuring principles "nesting", "ordering of parallel subprograms", and "use of subunits" yield a conceptual framework within which the structure of programs can be characterized in a qualitative manner. Furthermore we show that the program structures which can be characterized by the three structuring principles can be realized in Ada. The use of a real programming language seems to be more realistic than the use of an enriched Pascal as in [Ten 82]. Tennent uses for the two examples in his paper two different extensions of Pascal.

Section 2 introduces three principles of program structuring. In sections 3 and 4 the elements of Ada for program structuring are explained and Ada versions for quicksort and the banker's algorithm are given. These programs are compared with those of Hanson and Tennent. Conclusions are offered in section 5.

2 PRINCIPLES OF PROGRAM STRUCTURING

Before introducing the three principles of program structuring (for monolithic programs with the usual global visibility rule) we recall some properties of block structure for such programs.

Block structure generally allows the (textual) nesting of blocks.

There is one outermost block which contains all other blocks of the program. This outermost block is said to be on level 0. A block B immediately contained in a block A, which is on level n, is said to be on level n+1. We also say that B is on higher level than A. Thus the outermost block is on the lowest level.

Two blocks immediately contained in some declaration part are said to be parallel. They are on the same level.

There are two kinds of dependency between blocks:

d1) block B uses an entity (type, variable, etc) immediately declared in block A. A consequence of this is that B must be contained in A but not necessarily immediately.

d2) block A is called by block B, ie a call of A is contained in the statement part of B. A consequence of this is that A must be placed in such a manner that it is visible in the statement part of B; eg A cannot be contained in a block C contained in B.

In the following we define three principles of program structure for monolithic programs :

s1) the kind of nesting (flat/deep)

s2) the kind of ordering of parallel subprograms (alphabetical/logical)

s3) the use of subunits (yes/no)

With respect to these three principles of program structuring we say that there are different structures of a given program P if the blocks of P may be rearranged, ie we abstract from other structural properties and concentrate our view on the arrangements of the blocks.

It may be possible to change the arrangement of the blocks in a given program P in such a way that the new program P' has the same semantics ie shows the same behaviour as P. We then call P' a structural variant of P and vice versa. The kind of changes we are dealing with will be explained in the following paragraphs.

For all rearrangements we assume that they do not interfere with other structural properties of the respective program as eg the scope and visibility of entities. There always exist programs which allow the rearrangements discussed in the following. For other programs it may be necessary to transform them by means not discussed here (eg renaming of entities) in order to allow the rearrangement of blocks.

We do not deal with anonymous blocks because these are statements and thus not candidates for rearrangements.

The kind of nesting

We define a scale of nesting ranging from flat to deep. Flat nesting means that a subprogram is always declared at the lowest level possible, ie immeditately in the innermost block it depends on (dependency d1). Deep nesting means that a subprogram S is always declared at the highest level possible, ie immediately in the innermost block which contains all calls of S.

We do not deal with the case where a subprogram may be nested deeper if it is duplicated. Such a situation is depicted in figure 1. Subprogram A is called from B and C. If we do not want duplication it must at least be placed on the same level as B and C. This situation is shown in 1(a). If we duplicate A we can incorporate a copy of it in B and C. These two copies of A are then nested more deeply. This is depicted in 1(b).

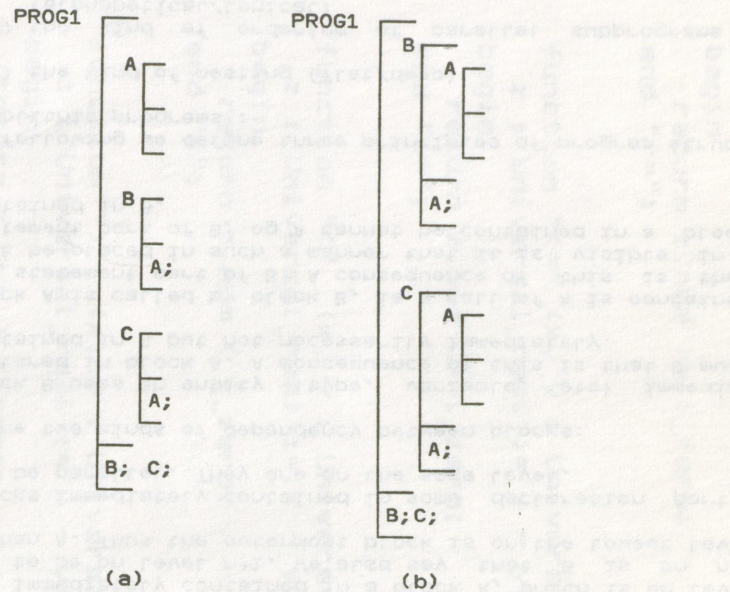


Figure 1. Deepest nesting may require duplication

Flat and deep characterize the end points of a scale of possibilities for structuring a program. Figure 2 shows an example for a program structure which is neither flat nor deep.

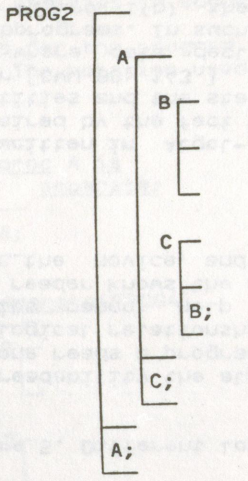


Figure 2. Intermediate program structure

Since in PROG2 procedure B is not used in the body of A directly its declaration could be moved "upwards" into the procedure C. If procedure B does not depend on A, i.e. does not use any entities declared immediately in A, its declaration could also be moved "downwards" into the declaration part of PROG2. Figure 3 shows the resulting three program structures of PROG2 in the form of tree diagrams, where an downward edge means "textual inclusion".

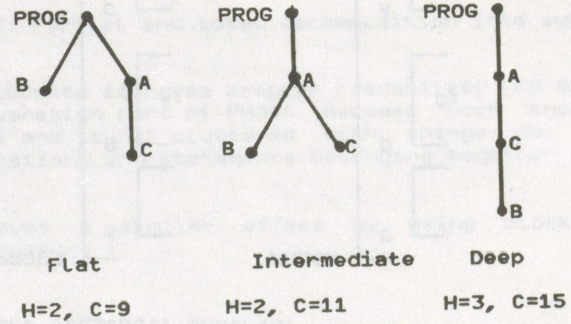


Figure 3. Three structural variants of PROG2

Ordering of Parallel Subprograms

If a declaration part contains immediately n subprograms these can be arranged in n! different arrangements. Alphabetical and logical ordering are two principles to arrange such parallel subprograms. By logical ordering we mean an arrangement which is consistent with the rule "declaration before use (DBU)". Alphabetical and logical ordering must not necessarily lead to different arrangements since by renaming we can always make the results identical. Since we believe in the advantages of mnemonically good identifiers we exclude such arbitrary renaming in the following discussion.

The alphabetical order is especially important in flat program structures where the number of subprograms in one declaration part tends to be larger than in deeply structured programs.

The alphabetical order may conflict with the usual DBU-rule. In languages containing a forward construct as eg Pascal [ISO 82] or which allow the separation of specification and body as eg Ada [Ref 83] this conflict can be easily resolved by first declaring the conflicting subprograms as forward and then list the bodys in alphabetical order.

The best scheme would perhaps be to declare <u>all</u> subprograms as forward and arrange these forward declarations already in alphabetical order. This sequence of forward declarations acts then as a sort of directory of the declared parallel subprograms. Such a scheme was succesfully used by the author in an interpreter for the program generation language PROGES [BWW 76: 5 .. 18]. This interpreter consists of a one pass compiler from PROGES in a low level intermediate language similar to P-code and an interpreter for this intermediate language both written in Burroughs Extended Algol. The compiler used recursive descent and contained for each nonterminal a Boolean function. Apart from these syntactic subprograms it contained a number of auxiliary subprograms. These two sets of subprograms where arranged as two different parts. The structure of this compiler is depicted in Figure 4.

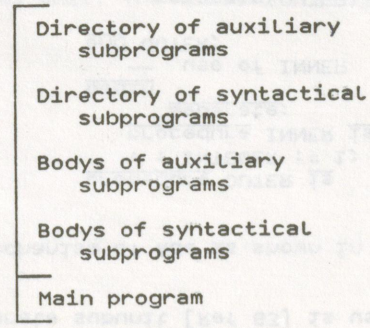


Figure 4. Coarse program structure of a compiler

Whereas the alphabetical order defines exactly one of the n! possibilities the logical order may characterize a subset of all permutations. Figure 5 shows an example where two different arrangements are logically ordered.

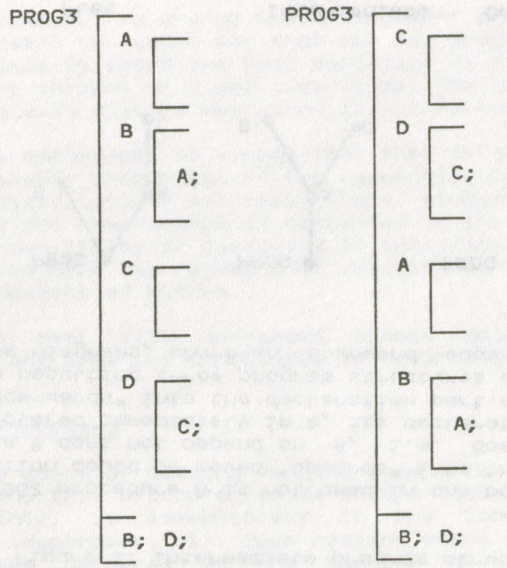


Figure 5. Different logical orderings

For the reason of readability the alphabetical order is to be preferred. If someone reads a program for the first time he does not know internal logical relationships. As a consequence these logical relationships cannot help him in understanding the program. Since any reader knows the alphabetical order it will be a great help for the novice and the casual reader of a program.

Use of subunits

In large programs written in Algol- or Pascal-like languages readability is impaired by the fact that for outer blocks the declarations of entities and the statements which use them are far from each other [CWW 80: 143]. This is especially the case in Pascal programs where data declarations must precede the declarations of subprograms. In such a program with a structure similar to that in figure 1(b) there can easily be several thousand lines of code between the data declarations and statements of the main program.

This unsatisfactory situation can be avoided if the concept of a

textually separate subunit [Ref 83] is used.

The subunit mechanism of Ada is shown in figure 6.

```
procedure OUTER is
   V : INTEGER := 1;
   procedure INNER is
      separate;
begin
   --  use of INNER
end OUTER;

     separate(OUTER)
procedure INNER is
   -- body of INNER
   -- use of V possible
end  INNER;
```

Figure 6. Subunit mechanism of Ada

A program unit which is declared in the outermost declaration part of another program unit can be declared as a subunit. This means that only the specification (eg procedure heading) is given and the body is replaced by the keyword separate. In Ada this construct is called a body stub. In figure 6 the procedure OUTER contains a body stub for the procedure INNER.

The body stub contains all information which is necessary to use this unit and it contains only this information.

The complete body of the subunit constitutes a textually separate program unit. It begins with a clause which mentions the containing unit ("separate(OUTER)" in figure 6). For procedures the heading is repeated, ie a complete procedure declaration is given (cf. figure 6). For other program units the subunit does not always contain the heading. This has been described elsewhere [Win 82]. Inside a subunit the same entities are visible as at the point of the corresponding body stub, eg the variable V which is declared in OUTER may be used inside the body of INNER as is indicated in figure 6.

Parent unit (procedure OUTER in figure 6) and subunit may be compiled separately where the parent unit must be compiled before the subunit. When compiling the subunit the compiler must perform all checks in the same way as if the subunit stood at the place of the body stub. If in figure 6 the variable V is used in the subunit INNER the compiler will check that the use is consistent with the definition in OUTER. This is the reason that OUTER must be compiled before INNER.

The use of subunits may interfere with readability if heavy use of global entities is made in the subunit and the program is read in a "bottom-up manner". The use of global entities is often considered harmful [SW 73]. Thus we do not recommend the

use of many global entities in subunits.

The use of subunits for the program structure of figure 2 is shown in figure 7. The notation used is essentially that of Ada.

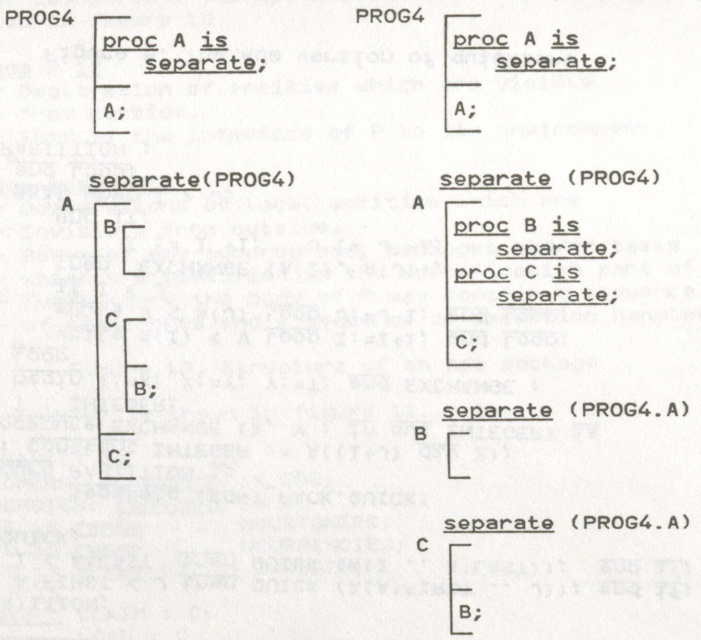
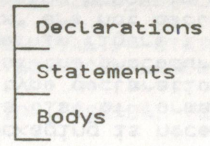


Figure 7. Partial and total decomposition into subunits

The use of subunits improves program readability [FG 82: 87] in that the declaration part of PROG4 becomes much shorter than that of PROG2 and is not cluttered with things not belonging there. Declarations and statements belonging together can be put together.

Tennent achieves a similar effect by using blocks of the following structure :



In this block the declaration part contains only the specification of locally declared subprograms. A similar effect is also achieved by the "refinements" of ELAN [HJJ 79: 23] and B [Geu 82: 55]. None of these three proposals does allow for separate compilation in the sense of Ada.

The technique of subunits can be used with all four combinations of the previously discussed principles "nesting" and "ordering". Thus we have eight different distinguished kinds of program structure and many more intermediate forms.

The effect of the different forms of program structuring can be seen at the concrete programs in the next two sections of the paper. Compared with nesting and ordering the use of subunits has the greatest influence on program readability because declarations and applications of entities come together as close as possible. We use Ada to show this effect.

3 QUICKSORT

The subunit concept leads to a similar structure as given in the examples of [Ten 82] where the bodies of local subprograms are placed behind the statement part of the block. The main difference is that in Ada the body of a subunit is a textually separate program unit. The quicksort program in Ada is depicted in figure 8.

```
package SORT_PACK is
   type VECTOR is array(INTEGER range <>) of INTEGER;
   procedure SORT (A : in out VECTOR);
end  SORT_PACK;
package body SORT_PACK is
   procedure QUICK (A : in out VECTOR) is
      separate;
   procedure SORT (A : in out VECTOR) is
      begin  if A'FIRST < A'LAST
             then  QUICK(A);  end if;
      end  SORT;
end SORT_PACK ;

          separate (SORT_PACK)
procedure QUICK (A : in out VECTOR) is
   I : INTEGER := A'FIRST;
   J : INTEGER := A'LAST;
   procedure PARTITION is separate;
begin
   PARTITION;
   if A'FIRST < J then QUICK (A(A'FIRST .. J)); end if;
   if I < A'LAST  then QUICK (A(I .. A'LAST));  end if;
end  QUICK;

          separate (SORT_PACK.QUICK)
procedure PARTITION is
   V : constant INTEGER := A((I+J) div 2);
   procedure EXCHANGE (X, Y : in out INTEGER) is
      T : INTEGER;
      begin T:=X; X:=Y; Y:=T; end EXCHANGE ;
begin loop
         while A(I) < V loop I:=I+1; end loop;
         while V < A(J) loop J:=J-1; end loop;
         if I ≤ J
         then  EXCHANGE (A(I), A(J));
               I := I +1;    J := J -1;
         end if;
      exit when I > J;
      end loop;
end  PARTITION ;
```

Figure 8. The Ada version of quicksort

The overall structure of the program in figure 8 is depicted in figure 9.

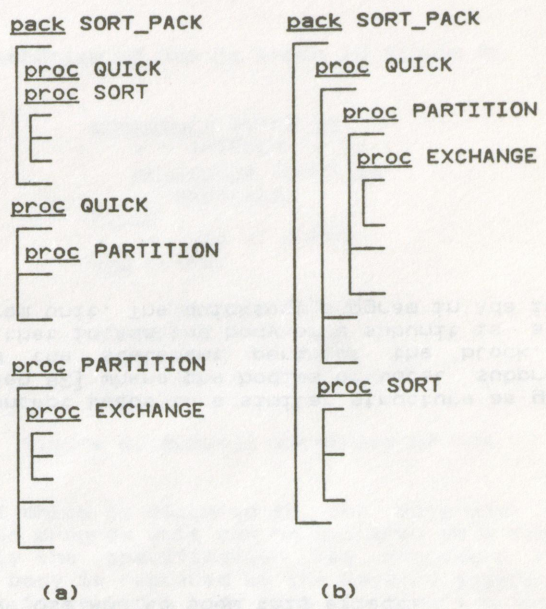


Figure 9. Coarse structure of the Ada quicksort

Part (a) shows the textual and part (b) the logical structure of the program in figure 8. The logical structure indicates that the program is deeply nested but due to the subunit concept of Ada this logical nesting does not lead automatically to textual nesting and thus to textually large program units.

The algorithm is similar to that of Wirth [Wir 76: 79] with some minor modifications. The main difference is that the structure of the program is very similar to that of Hanson's quicksort in order to make a comparison easier. The procedure QUICK is generally applicable if INTEGER'FIRST < A'FIRST and A'LAST < INTEGER'LAST.

The procedure SORT is embedded in a package which is called SORT_PACK. This packaging is necessary because an anonymous type cannot be used in a list of formal parameters. Therefore we have to put a suitable type declaration (for VECTOR) together with the specification of the procedure SORT in the specification part of the package. In figure 1 of Tennent's paper the types "vector" and "index" are not declared. In Pascal it could only be done in an enclosing block in which the declaration of the procedure SORT had also to be contained.

4 BANKER'S ALGORITHM

For the banker's algorithm we use the package concept to a higher degree than in the quicksort example. An Ada package may consist of two parts : the specification part and the body. This is depicted in figure 10.

```
package P is
   -- Declaration of entities which are visible
   -- from outside.
   -- This is the interface of P to its environment.
end  P;
package body P is
   -- Declarations of local entities which are
   -- invisible from outside.
   -- Bodys of any subprograms, packages and/or tasks
   -- which are specified in the specification part of P.
   -- Furthermore the body of P may contain a sequence
   -- of statements and a sequence of exception handlers.
end  P;
```

Figure 10. Structure of an Ada package

The "Ada Banker" is shown in figure 11.

```
package BANKER is
   NCUSTOMERS : constant := 200;
   NCURRENCIES: constant := 5;
   type B is range  1 .. NCUSTOMERS;
   type D is range  1 .. NCURRENCIES;
   type C is array (D) of INTEGER;
   type H is
      record CLAIM : C;
             LOAN : C;
             COMPLETED : BOOLEAN;
      end record;
   type TRANS is array (B) of H;
   type S is
      record TRANSACTIONS : TRANS;
             CAPITAL : C;
             CASH    : C;
      end record;

   function SAFE (CURRENTSTATE : S) return BOOLEAN;
end  BANKER;    -- end of specification part

package body BANKER is
   function ALL_TRANSACTIONS_COMPLETED(STATE : S) return BOOLEAN;
   function COMPLETION_POSSIBLE (CLAIM, CASH : C) return BOOLEAN;
   procedure COMPLETE_TRANSACTIONS (STATE : in out S);
   procedure RETURN_LOAN (LOAN : C; CASH : in out C);
```



```
   -- now all bodies follow in alphabetical order
   function ALL_TRANSACTIONS_COMPLETED(STATE : S) return BOOLEAN is
   begin
      return STATE.CAPITAL /= STATE.CASH;
   end  ALL_TRANSACTIONS_COMPLETED;

   function COMPLETION_POSSIBLE (CLAIM, CASH : C) return BOOLEAN is
   begin
      for CURRENCY in CLAIM'RANGE
      loop if CLAIM(CURRENCY) > CASH(CURRENCY)
           then return FALSE; end if;
      end loop;
      return TRUE;
   end  COMPLETION_POSSIBLE;

   procedure COMPLETE_TRANSACTIONS(STATE : in out S) is
      PROGRESS : BOOLEAN;
   begin
      loop
       PROGRESS := FALSE;
       for CUSTOMER in S.TRANSACTIONS'RANGE
       loop declare
               TRANS : H renames STATE.TRANSACTIONS(CUSTOMER);
            begin
               if not TRANS.COMPLETED
               then if COMPLETION_POSSIBLE(TRANS.CLAIM, STATE.CASH)
                    then RETURN_LOAN(TRANS.LOAN, STATE.CASH);
                         TRANS.COMPLETED := TRUE;
                         PROGRESS := TRUE;
                    end if;
               end if;
            end;
       end loop;
       exit when not PROGRESS;
      end loop;
   end COMPLETE_TRANSACTIONS;

   procedure RETURN_LOAN (LOAN : C; CASH : in out C) is
   begin
      for CURRENCY in CASH'RANGE
      loop
         CASH(CURRENCY) := CASH(CURRENCY) + LOAN(CURRENCY);
      end loop;
   end  RETURN_LOAN;

   function SAFE (CURRENT_STATE : S) return BOOLEAN is
      STATE : S := CURRENT_STATE;
   begin
      COMPLETE_TRANSACTIONS (STATE);
      return ALL_TRANSACTIONS_COMPLETED (STATE);
   end SAFE;
end  BANKER;
```

Figure 11. The banker's algorithm in Ada

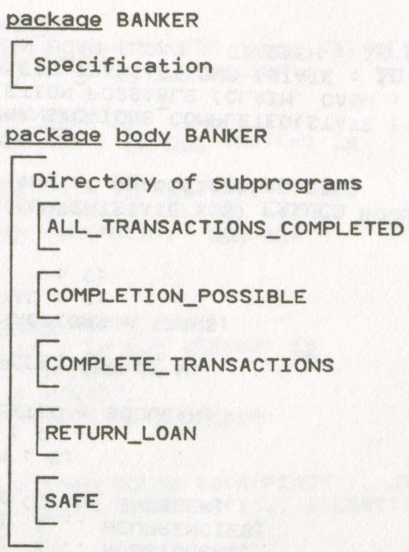


Figure 12. Coarse structure of the "Ada Banker"

In Ada an entity must first be declared before it can be used. If some parallel subprograms whose logical order differs from their alphabetical order should be arranged in alphabetical order the specification must be separated from the body. This has been done in figure 10 in the manner discussed in section 2. The algorithm is essentially the same as in [Han 81]. Only the function COMPLETION_POSSIBLE has been formulated in a more efficient manner and is quite similar to the version of Brinch-Hansen [Bri 73: 47]. The structure of the program in figure 11 can be characterized by

(Nesting = Flat, Ordering = Alpha, Subunits = No)

The overall structure is depicted in figure 12.

5 CONCLUSION

Using the principles defined in section 2 the examples in the three papers by Hanson, Tennent and Winkler can now be characterized as follows :

		Struct	Order	Subunit
Hanson	Fig.1 (Quick)	deep	/	no
	Fig.2 (Banker)	deep	alpha	no
	Fig.4 (Quick)	flat	DAU	no
	Fig.5 (Banker)	flat	alpha	no
Tennent	Fig.1 (Quick)	deep	/	no
	Fig.2 (Banker)	deep	stoch	no
Winkler	Fig.8 (Quick)	deep	alpha	yes
	Fig.11(Banker)	flat	alpha	no

/ : not applicable
DAU : declaration after use
stoch: neither DBU nor DAU

Table 1. Characterization of program examples

Table 1 indicates that the different program structures preferred by Hanson and Tennent can both be realized in Ada. The Ada version of quicksort (cf figure 8) is deeply nested and has thus a similar structure as the example in the paper of Tennent. The Ada version of the banker's algorithm (cf figure 11) has a flat structure. Therefore it resembles more the examples preferred by Hanson.

The module concept recommended by Hanson is essentially the same as the package concept of Ada. The where-clause of Tennent is logically equivalent to the subunit mechanism of Ada. The declaration blocks used by Tennent can be realized by Ada packages since entities local to a package body are not visible outside the package (cf figure 10). Thus the program structures recommended by Hanson and Tennent can all be realized in Ada. The important concepts which give Ada this flexibility are the package concept, the separation of specification and body, and the subunit concept.

Another important difference between the two kinds of program structure is one of visibility. In the quicksort program the visibility of the subprogram names is limited according to the need-to-know principle [Par 72]. But as a consequence of the global view inside of nested program units, which was incorporated in nearly all higher programming languages since Algol 60, the visibility of local entities is not limited according to the need-to-know principle. The procedure EXCHANGE eg sees the local variables I, J and V declared in the function PARTITION but it need not see them. The global view was also

criticized by Shaw and Wulf [SW 73] and Levy [Lev 82].

In the banker's program the situation is just the other way round : the visibility of the subprogram names is not limited whereas the visibility of local entities is.

It seems to be more error prone if local entities can be manipulated in an illegal way than if subprograms can be invoked illegally. Thus there seems to be a certain preference for structures similar to those of Hanson and the Ada banker.

ACKNOWLEDGEMENT.
The author thanks C.Stoffel and P.Wehrum for very helpful comments on earlier versions of the paper.

6 REFERENCES

Bri 73 Brinch Hansen, Per
Operating System Principles
Prentice Hall Inc., Englewood Cliffs, N.J. 1973.

BWW 76 Wettstein, H.; Becker-Weimann, K.; Winkler, J.F.H.; Wosnitza, H.
Ein modernes, modulares Betriebssystem für Prozeßrechner und seine Generierung.
PDV-E71, Gesellschaft für Kernforschung, Karlsruhe, Juni 1976.

CWW 80 Clarke, Lori A.; Wileden, Jack C.; Wolf, Alexander L.
Nesting in Ada Programs is for the Birds
SIGPLAN Notices 15,11(1980)139..145.

FG 82 Feuer, Alan R.; Gehani, Narain H.
A Comparison of the Programming Languages C and PASCAL
Computing Surveys 14,1(1982)73..92.

Geu 82 Geurts, Leo
An Overview of the B Programming Language or B Without Tears
SIGPLAN Notices 17,12(1982)49..58.

Han 81 Hanson, David R.
Is Block Structure Necessary ?
Software-Practice + Experience 11(1981)853..866.

HJJ 79 Hommel, Günter; Jäckel, Joachim; Jähnichen, Stefan; Kleine, Karl; Koch, Wilfried; Koster, Kees
ELAN-Sprachbeschreibung
Akademische Verlagsgesellschaft, Wiesbaden 1979.

ISO 82 ISO - International Organization for Standardization
Programming Languages - PASCAL, ISO/DIS 7185, 1982-08-12.

Lev 82 Levy, Eric B.
The Case against Pascal as a Teaching Tool
SIGPLAN Notices 17,11(1982)39..41.

Par 72 Parnas, D.L.
On the Criteria to be used in Decomposing Systems into Modules
CACM 15,12(1972)1053 .. 1058.

Ref 83 Reference Manual for the Ada Programming Language.
ANSI/MIL-STD 1815A.
United States Department of Defense, January 1983.

SW 73 Wulf, W.; Shaw, Mary
Global Variable Considered Harmful
SIGPLAN Notices 8,2(1973)28-34.

Ten 82 Tennent, R.D.
Two Examples of Block Structuring
Software - Practice + Experience 12(1982)385..392.

Win 82 Winkler, J.F.H.
Ada: the new concepts
Elektron. Rechenanlagen 24,4(1982)175..186 (in german).

Wir 76 Wirth, Niklaus
Algorithms + Data = Programs
Prentice-Hall,Inc., Englewood Cliffs, N.J. 1976.

Ada® LETTERS

A Bimonthly Publication of AdaTEC,
the SIGPLAN Technical Committee on Ada

VOLUME III NUMBER 6 MAY, JUNE 1984