PICA — A graphical program development tool*

AIMO A. TÖRN

Åbo Akademi, Dep. Comput. Sci., DataCity
SF—20520 ÅBO, Finland

Abstract

A technique and a tool PICA for rigorous program development with flowcharts is presented. This technique uses stuctured program flowcharts extended with assertion nodes containing program variable names and assertions about their values. An assertion node is connected to or from a statement node depending on if it represents a pre-condition or a post-condition. A tool for convenient use of the technique has been implemented as an Add-On to the Design software of Meta Software on a Macintosh II. The feasibility of using PICA is demonstrated by developing an algorithm for a small non-trivial programming task. The incentive for presenting the PICA technique is to create broader interest for rigorous programming methods by presenting one technique applicable to program development using flowcharts.

Index Terms—Automatic programming, computer-aided design, graphics, flowcharting, program correctness, rigorous programming, software design.

1. Introduction

There exists a rather extensive litterature on rigorous program development only to mention the text books [Jones 1980, 1986; Gries 1981; Reynolds 1981; Bjørner and Jones 1982; Backhouse 1986]. However, at large rigorous methods are still rather seldom used by programmers in practice. Reasons for this may be that rigorous methods require additional knowledge from their users, that the methods are deemed as labourious and thus unpractical, and that they are not easily integrated with generally used informal program development methods.

In order for a technique to be accepted by a larger group the technique should not be more formal than necessary and should be naturally integrateable with some well known informal program development method. Our choice of informal method upon which to build the rigorous tool is the graphical flowchart based method used

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

in HOS [Hamilton and Zeldin 1976] and there adjusted to structured programming practice. For rigorous program development using plain text several techniques exist, however, since Floyd presented his technique [Floyd 1967] techniques for flowcharting tools seem to have rendered very little interest. Despite the fact that flowcharts are more expressive and allow easier screening because of their ability to efficiently use the possibilities of the two-dimensional medium (paper, screen) used by man, their use have decreased over the years. One explanation of this might be the shift from off-line to on-line program design using text editors, which normally do not support graphical representation. The programmer has thus been forced to choose between *on-line working — plain text representation* or *off-line working — graphical representation.* The effect has been further increased by the same change towards plain text representation which can be noticed in the programming literature.

The rapidly increasing number of installed workstations with graphics makes the technique *on-line working — graphical representation* available to an increasing number of programmers. There seems to be a rising interest today in using graphical representation to increase the quality (e.g., readability and correctness) of intermediate and final program designs by the proponents of rigorous program development [Buhr *et al* 1989]. We will here demonstrate the use of a graphical tool PICA (Program—Information Charts with Assertions) supporting rigorous program development. The tool has been implemented as an Add-On to the Design software of Meta Software. In PICA pre-and post-conditions are added to the flowcharts as explicite graph elements showing the program variable names together with assertions on their values [Törn 1980, 1981].

The PICA technique will be explained in Sec. 2 using a trivial programming task. In Sec. 3 the PICA tool is used to derive a program for *The Longest Upsequence* problem [Gries 1981].

2. The PICA Technique and Tool

We first discuss formal program development and then illustrate the PICA technique and tool using a simple programming task.

2.1. Formal Program Development. Programming aims at establishing the result condition R. Correctness of a program S thus means both finding S and verifying that R is true when the program stops (partial correctness) and verifying that the program will always stop (correctness).

Normally developing S and verifying R can be made only if some precondition Q is valid when the program starts. For instance, when a program for computing the square root of a real number is developed it is naturally assumed that the real number is greater than or equal to zero. However, the programmer cannot be sure that the program will always be used as intended and good programming practice therefore means that the program should address also the complementary case.

A well designed program should therefore contain an initial part that decides whether Q is valid or not and produces some "natural" result (e.g. an error message) for $\neg Q$ represented by the truthness of an exception condition R_e. Correctness further requires that Q and $\neg Q$ are evaluable and therefore a pre-initial part S_0 of the program (possibly containing inputs, must be written that secures this. This is

in accordance with Floyd's PROMP-READ-CHECK-ECHO implementing the idea that each component of a program should be protected from input for which that component was not designed [Floyd 1979]).

A program specified in this way can be said to be *properly specified* because for each possible pre-condition a specification of the corresponding wanted result exists. A properly specified program thus has the following general appearance

$$S_0; S';$$

where

$$S': \textbf{if} \neg Q \rightarrow S_e \;\blacksquare\; Q \rightarrow S \;\textbf{fi}$$

2.2. The Flowchart Technique. The flowchart elements used to describe *sequence, choice, iteration, refinement* and *assertions* are shown in Fig. 1. The elements are those used in HOS, with exception for the element of choice here represented structured in the same way as the element for iteration. Assertions are represented by dotted boxes divided into an upper and a lower part. The upper part contains the name of a variable, the lower part a predicate on that variable. The implied assertion is that the predicate is true.

Several assertion boxes may be connected by the logical operators *and, or* and $\Rightarrow$. A dotted arrow pointing from a statement box to an assertion box means that the assertion is valid after the execution of the statements in the statement box. A dotted arrow from an assertion box to a statement box shows that the assertion in the assertion box is valid when the computation reaches the statement box.

The flowchart in PICA notation corresponding to a properly specified program is shown in Fig. 2. The usual flowchart notation for the conditions valid at the branches of the *if*-statement (case statement) is used.

For proving that the program is correct we have to find S_e, S and to assert the result conditions in the PICA graph. For iterations it must also be proved that they are finite. The proof procedure consists of recursive applications of refinements of S, R and correctness proofs until such a program detail is reached that every step is sufficiently convincingly proved.

2.2. A Simple Programming Task. The result of using PICA for designing a program for adding the elements of a vector $x_1, \ldots, x_n$ is shown in the Appendix A. Some details of the PICA tool is also explained in the "text pages".

First a crude design is made. If a statement box must be refined a new flowchart page may be opened. On this child page an empty box with the surrounding of the refined box from the parent page will be exposed. The details of the design may then be introduced into the empty box. For each flowchart page there is a corresponding text page which can be used to complement the design so that a complete documentation of the design is obtained.

The tool is used in a three stage procedure. First the flowcharts together with the comments on the text pages are produced. For flowcharting a palette is available from which the flowchart elements needed are chosen and copied to the flowchart page. The tool is implemented on a Macintosh II with big screen which admits to have the plaette, the flowchart page and the text page open simultaneously. When the flowchart is ready the proof stage is entered. Unproved statement boxes have

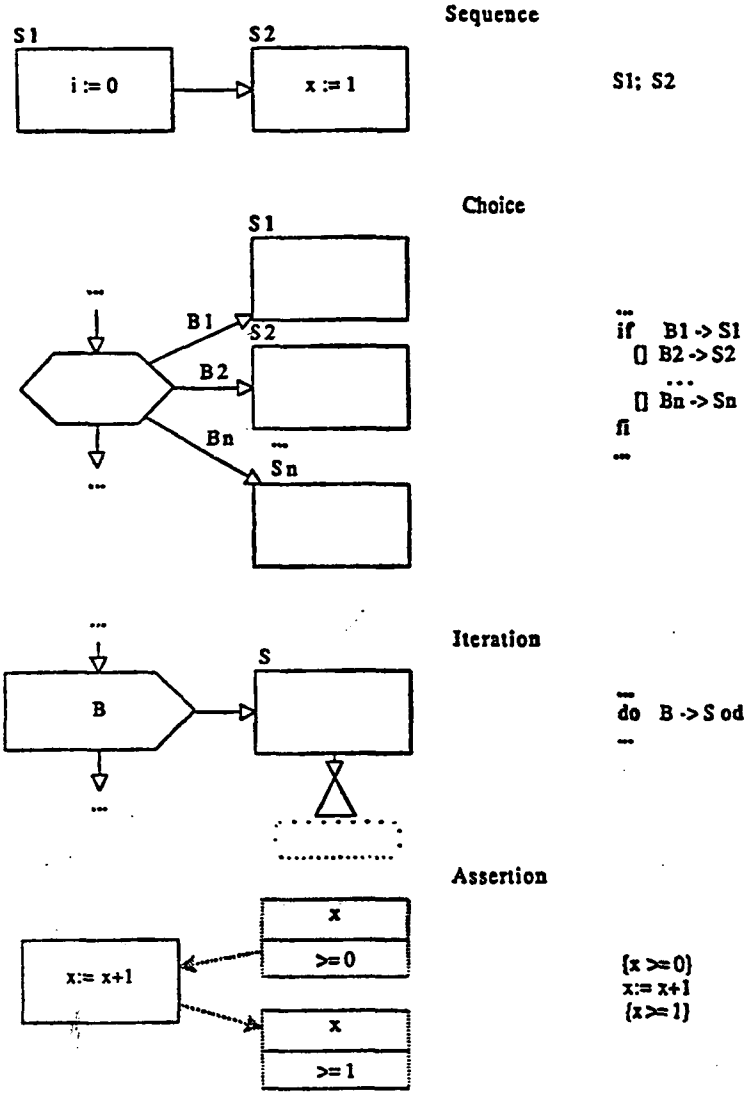


Fig. 1. PICA flowchart notations

thicker border lines than proved boxes. The PICA tool will keep track of the proof procedure so that the most refined parts have to be proved prior to cruder ones. There is no theorem proving facility available in the tool, i.e., the proofs are made informally by the user. In this activity the corresponding text page may be used to document the proofs. The tool will however check that pre- and post-conditions, and variants are existing where there must be such. It also reminds the user what

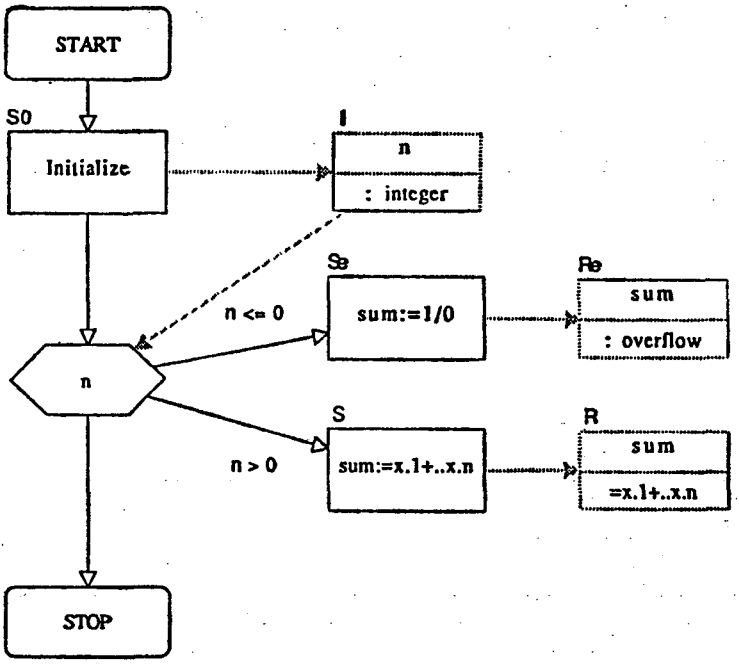


Fig. 2. General form of a program in PICA

have to be checked in order for a proof step to be complete. When the whole design has been proved correct the third stage which produces a skeleton program text may be entered. A printout from this stage is presented on the second text page of the design in Appendix A.

3. The Longest Upsequence Problem

The PICA tool will here be applied to the problem of designing an algorithm for finding the length of the longest upsequence *lup* (longest *up*) of a given vector $x_1, \ldots, x_n$, $n \geqq 0$. Based on the experiences of this some points are then discussed.

3.1. The Lenght of the Longest Upsequence. Let a *up* over x be defined as

$$up\colon (x_{(i)},\ldots,x_{(i+k)}),\ k\geqq 0,$$
$$(\ldots,x_{(i)},\ldots,x_{(i+1)},\ldots,x_{(i+k)},\ldots)=(x_1,\ldots,x_n),$$
$$x_{(i)}\leqq x_{(i+1)}\leqq\ldots\leqq x_{(i+k)}.$$

The resulting PICA design is shown in Appendix B.

3.2. Discussion. The development of an algorithm for the length of longest *up* starts with the division of the task to be performed into two cases, one of which is executed on each application of the algorithm. In order to be able to make refirement of the algorithm S knowledge about the problem to be solved is needed. This knowledge is presented as theorems about problem entities. The development then proceeds by successive refinement, using the knowledge contained in the theorems, and verification until such a detail is achieved that the algorithm can easily be coded using the target programming language.

4. Conclusions

The feasibility of using a specific rigorous program development technique PICA with flowcharts has been demonstrated by developing an algorithm for a non-trivial but small programming task. The PICA design is more easily screened and checked because of the greater freedom of flowchart representation. The technique is equivalent to several suggested techniques for plain text algorithm representation. The incentive for presenting the PICA technique is to create broader interest for rigorous programming methods by presenting one technique for those programmers who are proponents of flowcharting techniques for program development. In order to aid in using PICA a graphical tool supporting formal program development based on PICA is available. In addition to supporting the graphical representation and administering the proof procedure the tool is also able to automatically generate the equivalent plain text representation of the design including the assertions. This skeleton program can then be transformed into a compile ready version by further editing.

References

[Backhouse 1986] R. C. Backhouse, *Program construction and verification*, Prentice-Hall, Englewood Cliffs, N. J.

[Bjørner and Jones 1982] D. Bjørner and C. B. Jones, *Formal specification and software development*, Prentice-Hall, Englewood Cliffs, N. J.

[Buhr *et al* 1989] R. J. Buhr, G. M. Karem, C. J. Hayes and C. M. Woodside, *Software CAD: A revolutionary approach*, IEEE Trans. on Software Eng. 15, 235—249.

[Dershowits 1980] N. Dershowits, *The evolution of programs*, Technical Report UIUCDCS—R—80—1017, Department of Computer Science, Uni. of Illinois at Urbana-Campaign, 212 pp.

[Floyd 1967] R. W. Floyd, *Assigning meaning to programs*, In: Mathematical aspects of computer science 19, Amer. Math. Society, 19—32.

[Floyd 1979] R. W. Floyd, *The paradigms of programming*, Comm. of ACM 22, 455—460.

[Gries 1981] D. GRIES, *The science of programming*, Springer-Verlag, New York.
[Hamilton and Zeldin 1976] M. HAMILTON and S. ZELDIN, *Higher order software — A methodology for defining software*, IEEE Trans. Software Eng. SE—2, 9—32.
[Hoare 1969] C. A. R. HOARE, *An axiomatic bases for computer programming*, Comm. ACM 12, 576—580, 583.
[Hehner 1984] E. C. R. HEHNER, *The logic of programming*, Prentice-Hall, Englewood Cliffs, N. J.
[Jones 1980] C. B. JONES, *Software development, a rigorous approach*, Prentice-Hall, Englewood Cliffs, N. J.
[Jones 1986] C. B. JONES, *Systematic software development using VDM*, Prentice-Hall, Englewood Cliffs, N. J.
[Reynolds 1981] J. C. REYNOLDS, *The craft of programming*, Prentice-Hall, Englewood Cliffs, N. J.
[Törn 1980] A. TÖRN, *Structured programming using program flowcharts containing explicite representation of data including assertions*, Technical Report 10, Department of Computer Science, Åbo Akademi, Finland, 16 pp.
[Törn 1981] A. TÖRN, *PICA — A flowchart tool for structured programming supporting proving*, Technical Report 16, Department of Computer Science, Åbo Akademi, Finland, 17 pp.

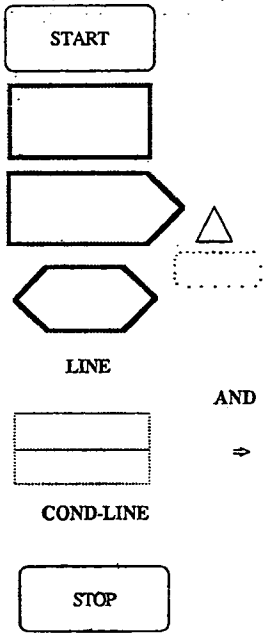
Vector Addition

A program for computing the sum of the elements of a vector x1+...+xn is to be designed. It is decided that n less than or equal to 0 is an exception and that the result produced in this case is an overflow condition.

The first part of the program is an initial part that guarantees that an integer is assigned to n. This is shown by the dotted arrow pointing from the box S0 to the box I.

From the condition box we have the two cases n<=0 and n>0. The box Se produces the exception and the box S the result for n>0.

The boxes have been copied from a palette similar to the one below.



LINE is chosen when connecting statement boxes and COND LINE when connecting a statement box and a assertion box.

There is a menu named FLOWCHART with the following options:

Open Palette

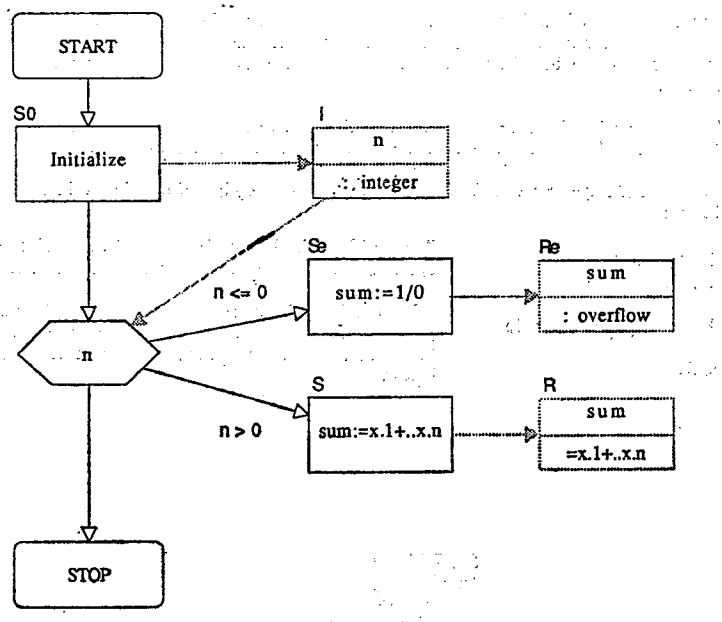
Open Text Page

Name Node
Fill in Guard
Refine Box

Set Obj. Horisontal
Set Obj. Vertical

Quit

We choose the box S and use the option Refine Box from the menu. This will produce a new flowchart page like the one on the next page but with initially empty inner part. Below S is initially only the box pointed to from the border.


START
S0
Initialize
I
n
: integer
n
n <= 0
Se
sum:=1/0
Re
sum
: overflow
n > 0
S
sum:=x.1+..x.n
R
sum
=x.1+..x.n
STOP

The refinement of the box S on the parent page is shown here. In the box S1 the initialization for the iteration is made and this makes the invariant P valid. The guard is given in box S2 and the variant n+1-i is shown at the end-of-iteration symbol.

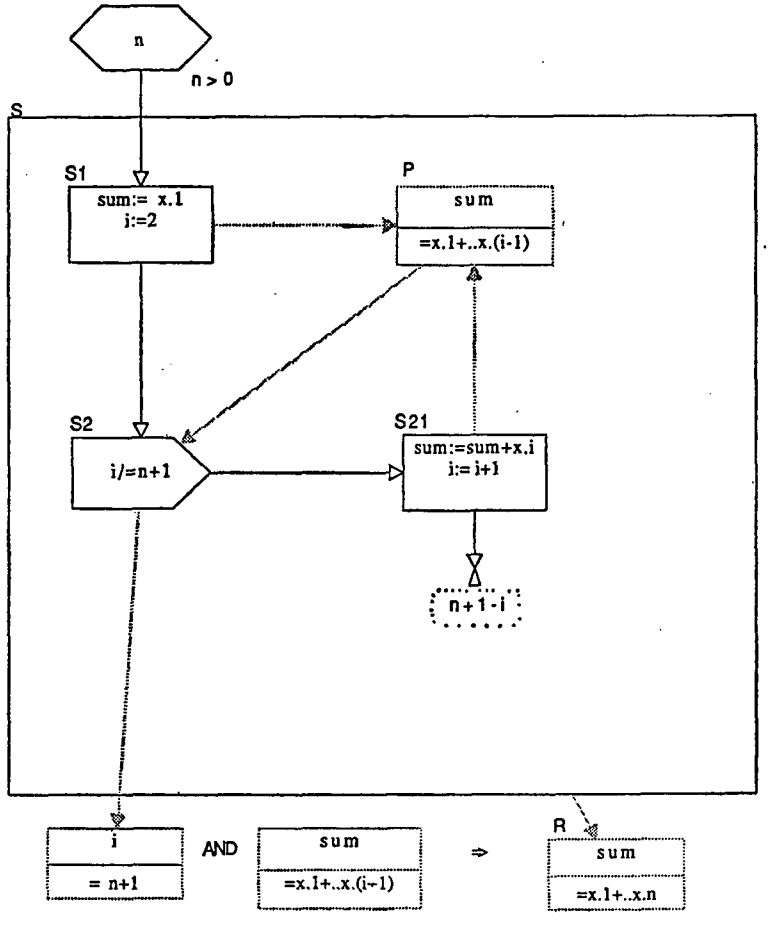
The iteration box is connected to the assertion box below S which shows the falsification of the guard and P. These together give the postcondition R.

In the proof stage a menu PROOF is used with the options shown in the box below:

Prove node	The proof starts by proving S1 and S21 and then S2. It will be checked that the variant box has text.
Unprove node	
Print flowchart on file	After proving S2 the remaining boxes on the parent page may be proved. After this we may use the Print-flowchart-on-file option. The result is shown below.
Quit	

```
S0: Initialize
{n : integer}
if n <= 0 ->
      Se: sum:=1/0
      {sum : overflow}
 [] n > 0 ->
      S: sum:=x.1+..x.n
      {sum =x.1+..x.n}
fi


S:
S1: sum:= x.1
i:=2
{Invariant: {sum =x.1+..x.(i-1)}
 Variant: n+1-i}
do i/=n+1 ->
      S21: sum:=sum+x.i
i:= i+1
od
{i = n+1 and sum =x.1+..x.(i-1)}
=>{sum =x.1+..x.n}
```


n
n > 0
S
S1
sum:= x.1
i:=2
P
sum
=x.1+..x.(i-1)
S2
i/=n+1
S21
sum:=sum+x.i
i:= i+1
n+1-i
i
= n+1
AND
sum
=x.1+..x.(i-1)
⇒
R
sum
=x.1+..x.n

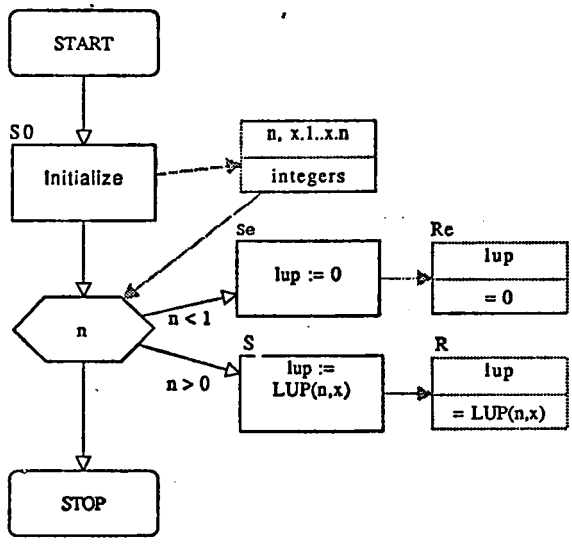
The longest upsequence problem

The length of the longest upsequence of elements given a vector x.1..x.n is to be determined. We use the notation LUP(n,x) for this number. The result of our algorithm is to be stored into the variable lup. Assume further that it is decided that the result for n<1 shall be 0.

A crude design of our algorithm is given in the page to the right. The following variables are used:

integer:
- n = number of elements in the vector
- x.1..x.n = vextor containing the n integers
- lup = the variable to contain the result of the algorithm.

The algorithm consists of a conditional statement covering all values of n. For n<1 Se gives lup the value 0 as required, and for n>0 the statement S assigns the correct value to lup. The design is obviously correct providing that LUP(n,x) is computed correctly. The refinement of S is shown on the next two pages.


START
S0
Initialize
n, x.1..x.n
integers
Se
lup := 0
Re
lup
= 0
n
n < 1
S
n > 0
lup :=
LUP(n,x)
R
lup
= LUP(n,x)
STOP

In order to show how to compute LUP(n,x) we need to state some results. Let LSE(i,x) be the longest upsequence ending in x.i. Then

THEOREM 1: LUP(n,x) = max LSE(i,x), where I=[1,n]. Obvious.
i in I

Our problem has now been reduced to computing LSE(i,x). For LSE(i,x) the following is valid:

THEOREM 2: LSE(1,x) = 1 and LSE(i,x) ≤ i, i= 2,...,n. Obvious.

THEOREM 3: Let n≥ 2. Then

$$LSE(i,x) = 1 + \max_{j \text{ in } A} LSE(j,x),\ i = 2,\ldots, n \text{ if } A \neq \emptyset$$

and

$$LSE(i,x) = 1 \text{ if } A = \emptyset,$$

where

$$A = \{j \mid 1 \leq j \leq i-1 \text{ and } x.j \leq x.i\}.$$

PROOF: For all upsequences ending in x.j, 1≤ j ≤ i-1 for which x.j ≤ x.i the element x.i can be added giving an upsequence one element longer. If A = ø then x.i is the smallest element among x.1,...,x.i and therefore an upsequence ending in x.i consists of just the single element x.i, a sequence whose length is 1.

We use the vector e.1..e.n to store the results of computing LSE(i,x), i = 1..n.
The computation of e.i can then be performed as follows:

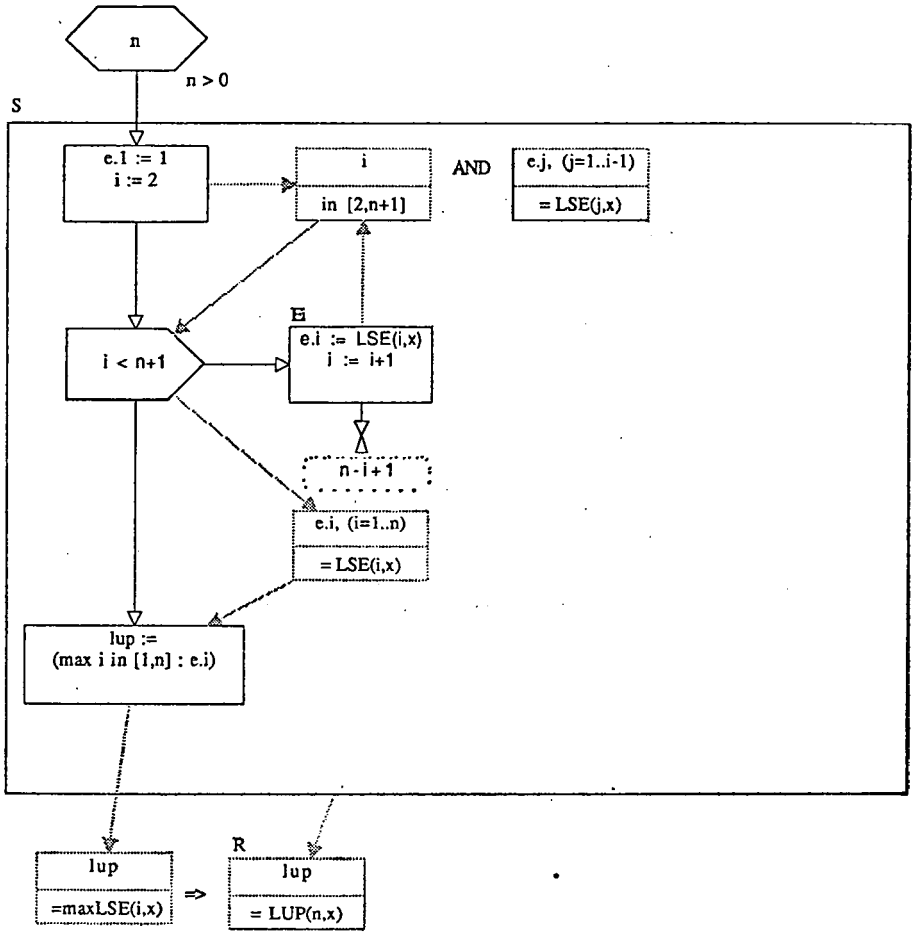
```
    i=1:
e.i  :=  1,

    i = 2..n:
e.i  :=  1  +  max  e.j,
              j in A
```

where A = { j | 1≤ j ≤i-1 and x.j ≤ x.i}. The design of computing the vector e.1.. e.n is shown to the right. The following variables are used:

```
integer:
  e.1..e.n  =  vector  to  store  LSE(i,x)
   i=    index
```

The only nontrivial task is now to compute e.i. The node Ei is therefore refined and its design is shown on the next page.


n
n > 0
S
e.1 := 1
i := 2
i
in [2,n+1]
AND
e.j, (j=1..i-1)
= LSE(j,x)
E
i < n+1
e.i := LSE(i,x)
i := i+1
n - i + 1
e.i, (i=1..n)
= LSE(i,x)
lup :=
(max i in [1,n] : e.i)
lup
=maxLSE(i,x)
=>
R
lup
= LUP(n,x)

The computation of e.i given e.1..e.(i-1) is shown to the right. The set B in the loop invariant is

$B=\{j \mid 1 \leq j \leq k-1 \text{ and } x.j \leq x.j\}$.

For k=i, B is equivalent to A.

The following variables are introduced:

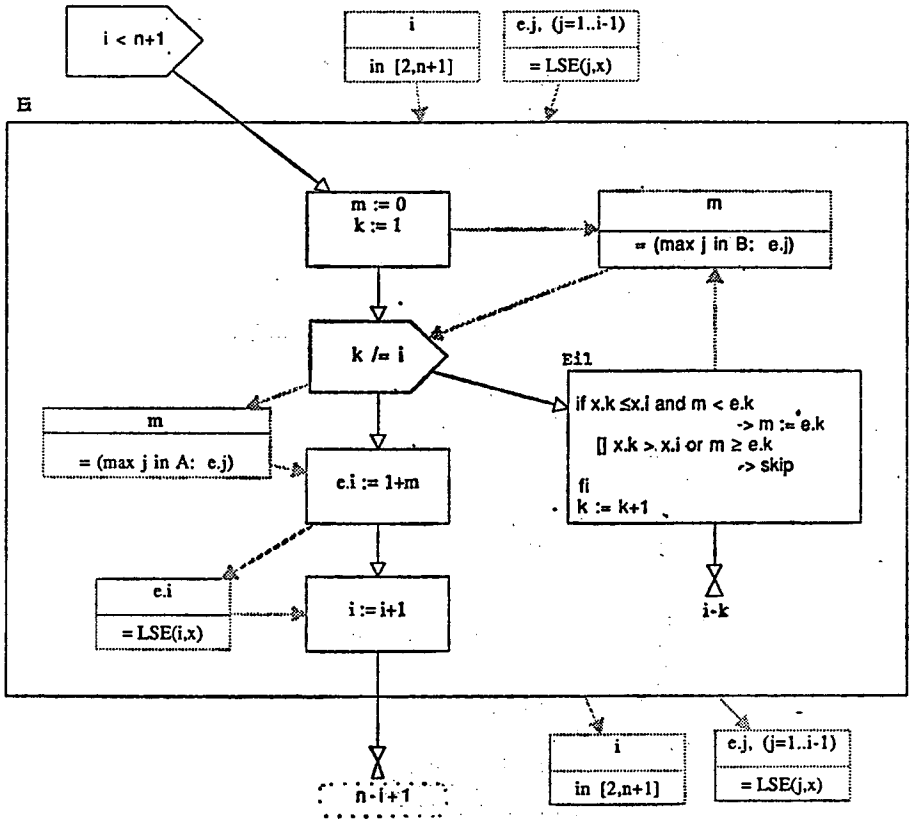
integer:
m: used to store (max j in B: e.j) for k = 1..i-1
k = index.

Note that in the box Ei1 code is written instead for showing the design in the form of a flowchart. By utilizing this feature the trivial parts can be written more condensed and only parts where formal reasoning is of help are shown as charts. This possibility makes it possible to use the tool in a very flexible way and may therefore suit different tastes of programming.

When the design is ready an explicite proof stage is entered by quitting the PICA Flowchart and choosing the PICA Proof Add On from the apple menu. An unproved box has a thicker border line. Proving the correctness of a node is done by klicking on the node. The prover then checks that subordinate nodes are proved and that the node has necessary pre- and postconditions.

It is supposed that by separating the design part and the proof part the user will have a better chance of finding an error than if both tasks are intervieńed.

When the design is proved correct one may choose the Print Flowchart on File option from the Poof menu. This will give a skeleton program consistent with the design given in the flowcharts. This program may then be completed by further editing. The skeleton program resulting from the design presented in this example is shown on the next page. An edited running version in Simula is also presented.


i < n+1
i
in [2,n+1]
e.j, (j=1..i-1)
= LSE(j,x)
Ei
m := 0
k := 1
m
= (max j in B: e.j)
k /= i
Ei1
if x.k ≤x.i and m < e.k
-> m := e.k
[] x.k > x.i or m ≥ e.k
-> skip
fi
k := k+1
m
= (max j in A: e.j)
e.i := 1+m
e.i
= LSE(i,x)
i := i+1
i-k
n-i+1
i
in [2,n+1]
e.j, (j=1..i-1)
= LSE(j,x)

```
S0: Initialize
{n, x.1..x.n  integers}
if n < 1 ->
      Se: lup := 0
          {lup  = 0}
 [] n > 0 ->
      S: lup := LUP(n,x)
         {lup  = LUP(n,x)}
fi


S:
e.1 := 1
i := 2
{Invariant: {i  in [2,n+1] and e.j, (j=1..i-1)  = LSE(j,x)}
 Variant: n-i+1}
do i < n+1 ->
      Ei:  e.i := LSE(i,x)
           i := i+1
od
{e.i, (i=1..n)  = LSE(i,x)}
lup :=
(max i in [1,n] : e.i)
{lup  =maxLSE(i,x)}
=>{lup  = LUP(n,x)}


Ei:
{i  in [2,n+1] and e.j, (j=1..i-1)  = LSE(j,x)}
m := 0
k := 1
{Invariant: {m  = (max j in B:  e.j) }
 Variant: i-k}
do k ≠ i ->
        Ei1:  if x.k ≤x.i and m < e.k
                                -> m := e.k
              [] x.k > x.i or m ≥ e.k
                                -> skip
              fi
              k := k+1
od
{m  = (max j in A:  e.j) }
e.i := l+m

{e.i  = LSE(i,x)}
i := i+1
{i  in [2,n+1] and e.j, (j=1..i-1)  = LSE(j,x)}
```

```
integer procedure lup (n, x); integer n; integer array x;                    comment
                                        ** lup = length longest upsequence of x.1...x.n;
⟨n, x.1...x.n integers⟩                                                          ;
begin
  integer procedure plup (n, x); ... ;
  if n lt 1  then                                                            comment
      Se; lup:=0;                    comment ⟨lup = 0⟩;
  if n gt 0  then                                                            comment;
      S ; lup:=plup (n,x);           comment ⟨lup = LUP(n,x)⟩;
end;
  integer procedure plup(n, x); integer n; integer array x;
  begin
    integer procedure plse (i,x,e); ... ;
    integer array e(1:n);
    integer i, max;
    e(1):=1;
    i   :=2;                                                                 comment
    ⟨Invariant: ⟨i in (2,n+1) and e.j, (j=1...i-1) = LSE(j,x)⟩
    Variant : n-1+1⟩                                                            ;
    while i lt n+1 do                                                        comment
        Ei; begin e(i):=plse (i,x,e);
                  i:=i+1
            end;                                                             comment
    ⟨e.i, (i=1, , n) = LSE(i, x)⟩                                               ;
    max:=e(1);
    for i := 2 step 1 until n do if max lt e(i) then max:=e(i);
    plup:= max;                                                              comment
    ⟨plup = max LSE(i,x)⟩                                                       ;
end;
    integer procedure plse (i,x,e); integer i; integer array x, e;
    begin                                                                    comment
      ⟨i in (2, n+1) and e.j, (j=1...i-1) = LSE(j,x)⟩                            ;
      integer m, k;
      m:=0;                                                                  comment
      k :=1;
      ⟨Invariant: ⟨m = (max j in B: e.j)⟩
      Variant : i-k⟩                                                             ;
      while k ne i do                                                        comment
          Eil; begin if x(k) le x(i) and m lt e(k)
                                  then m:=e(k);
                     k:=k+1
           end;                                                              comment
      ⟨m = (max j in A: e.j)⟩
      plse:=1+m;                                                             comment
      ⟨plse (i,x,e) = LSE(i,x)⟩                                                  ;
  end;
```