PROGRAM DEVELOPMENT AND DOCUMENTATION BY INFORMAL TRANSFORMATIONS AND DERIVATIONS

Giovanna Petrone   Luigi Petrone
Dipartimento di Informatica
Università di Torino, Italy

1. INTRODUCTION

The content of this paper(*) is the outcome of some teaching experience and a research effort in programming methodology. However, part of the terminology has been changed and some insight has been gained as an effect of the wider perspective set out by Scherlis and Scott in their quite recent IFIP 83 paper. Their overall analysis is assumed in this paper with some limitations of objectives; in fact we are principally interested in informal, but possibly rigorous, derivations of programs and in designing adequate supporting tools. Formal derivation of programs as well as formal correctness proofs are relevant to this paper, conceptually, in that they provide an insight, a deep insight indeed, into the correctness problem. In other words, by asserting the possibility of deriving a formal correctness proof we get the assurance that the informal, but nevertheless rigorous, proofs of program derivations are not without foundations and that they can become, anytime, formal and detailed if needed.

This lack of formalism can sometime induce a lack of precision and may not always be capable of preventing the introduction of programming errors. It is however difficult to guarantee a level of absolute precision within a methodology where the human contribution is substantial. But we dare confess that our primary objective is not absolute and foolproof program correctness, but

(*) Preparation of this paper supported in part by NSF grant MCS -81-03718 and by project ESPRIT 125, funded by EEC.

rather a methodology for program development, documentation and understanding, and we want to privilege an intuitive deep understanding of a program from the designer point of view rather than to wait for a distant, and yet to come, automatic support for the derivation process. We are aware that some confusion exists today in the field of programming methodology between formal treatment and rigorous treatment of programs. We devote an entire section to try to clarify this issue.

2. PROGRAMMING AS AN EVOLUTIONARY PROCESS

Programming, even real word programming, is an evolutionary process |Bauer 76, Balzer 81|. The final results of the programming process, the program texts, deprived of the insight that went into their conception, are too complex to be understandable and should not be used for the maintenance process. One might even say that the program texts are as relevant to the programming process as one accidental result of the addition process, say 26, is relevant to the understanding of the concept of the integer sum. In fact, programs do not stay immobile, they evolve and modify in synchronism with the social context in which they are exploited. Therefore, we are interested in the derivation process of a program more than in the program itself |Bauer 76, Scherlis and Scott 83 and also Petrone 83|.

The first known examples of program derivation techniques were the step-wise refinements of Dijkstra |72| and Wirth |71| and the program transformations of Burstall and Darlington |77|, Manna and Waldinger |79|. However, the general framework of the latter authors, perhaps biased by their cultural background of research in mechanical theorem proving, has been that of setting up a theory and a set of tools to provide an automated environment for deriving correct programs. This latter enterprise should not exclude the need to explore and experiment derivations of programs carried out by humans but no significant effort has been made in this direction. Certainly, a lack of convenient tools makes the job more difficult. But, probably, this is also a signal that the evolutionary nature of the programming process is generally seen, in the computing community, more as an evil than as a feature that can be of great help in explaining and documenting the design of a system.

3. FORMAL SPECIFICATIONS VERSUS RIGOROUS SPECIFICATIONS

It is claimed that software implementation should start from a formal specification language. The request for formality is based upon the statement that only a formal expression would allow a correctness proof to be carried over. This statement seems to imply a framework that happens to be too restrictive specially in view of the even stronger statement that only a formal treatment would allow a sound mathematical treatment of the programming process. It has been explicitly pointed out that this may not be true (see |P. Naur 82| and |De Millo 79|). The usual presentation of mathematical proofs appearing in technical journals even if often referred to as rigorous proofs (or sometimes, perhaps improperly, as formal proofs) is never carried using a formal predefined language provided of nonambiguous formal syntax. No mathematician would accept such a constraint in his everyday work.

Problems do arise because of the already mentioned confusion between formality and rigor . In classical mathematics formality and rigor are synonyms. Rigor only means that each concept is operationally defined according to sound rules accepted in a given discipline and notations are always defined in a way that excludes ambiguity of any sort. In Logic and even more so in Computer Science formality usually indicates that a language is defined with a precise syntax , given in BNF, and a precise semantics is given that makes mechanical treatment possible for instance by allowing a mechanical proof or an automatic checking . But to request that all reasoning about programs and in particular all derivations of programs be made in a formal language is probably an useless if not harmful approach.

Formality is a price we are willing to pay only if a substantial benefit from some advantageous mechanical processing is obtained, in return. In fact, the possibility of rigorous but not formal treatment allows for greater freedom in selecting the proper framework which happens to be the most convenient for each application field or each problem at hand. Moreover, it is doubtful that one might be able to define a unique specification language suitable for the many different application fields of the real world. The argument that since a programming language like Pascal is "Universal" we should be able to define a universal specification language seems not to apply completely. In fact, the universality of a programming language is achieved in each application by means of many layered levels of abstractions, each one level requiring what cannot be considered but as a heavy effort of implementation and coding.

4. SPECIFICATION: THE FIRST IMPLEMENTATION PHASE

Obviously, software problems do not have a unique specification. Moreover, as we will show on the example of the sorting problems, certain specifications make easy to derive certain implementations and prevent, or make it difficult to derive others.

In a certain sense, the specification language is a sort of programming language and the specification phase is the first, and not the least important, implementation phase. This viewpoint is shared by some authors. Bauer and Woessner |82| speak of some specifications as pre-algorithmic formulations of problems and Scherlis and Scott |83| say that the difference between specification and implementation is not qualitative but quantitative. To support the previous view we point out the following facts:

i) Specifications can often be interpreted as trivial algorithms, the so called British Museum algorithms. For instance the specification:

$$\forall x \ \exists n \mid x \geq 0 \Longrightarrow n^2 \leq x < (n+1)^2$$

can be interpreted as:

$$\text{given any } x \geq 0 \text{ find } n \mid n^2 \leqslant x < (n+1)^2$$

or more explicity if we know that such an n exists and $n \leq x$ it can be found by the trivial (descending) search program:

```
i := x;
while not $i^2 \leq x < (i+1)^2$ do
   i := i - 1;
```

ii) static specifications in the Hoare style can often be transformed into recursive definitions. See program synthesis by Darlington and Manna and their folding-unfolding techniques.
iii) specifications are often expressed in terms of concrete representations of the data structures of the problem and this requires a first implementation effort, a first level of commitment.
iv) specifications are often given in terms of notions that are defined constructively, i.e. recursively.
For instance, the summation sign definition: a = a+a+...+a looks certainly more esotheric than the simple constructive definition:

$$\sum_{n}^{m<n} a = 0 \qquad \sum^{n} = \sum^{n-1} + a_n$$

a But the latter recursive definition immediately leads to the corresponding recursive program and is fundamental to any iterative program such as

```
i := 0; s := 0;
while i  n do
   s := s + a|i|
   i := i+1
```

This appears more clearly for the usual <u>gcd</u> algorithm of two integers x, y. The specification is:

$$\forall\, x\ y\ \exists z \mid z/x \text{ and } z/y \text{ and } \forall w \mid w/x \text{ and } w/y \Longrightarrow z \geq w$$

A trivial interpretation would imply an exhaustive search. The usual algorithm can be derived only if the following two recursive equations are known:

$$\begin{aligned} gcd(x,y) &= gcd(x-y,y) && \text{if } x \geq y \\ gcd(x,y) &= x && \text{if } x=y \end{aligned}$$

More enlightening is the example of the sort programs. To derive a sort program one should first know what sorted sequence means. The usual definition is given in terms of a concrete representation of a sequence as an array:

(1) $$\forall\, i\ j \mid i<j \Rightarrow a[i] \leq a[j]$$

This definition, intended as a program specification, interpreted as a prescription leads to

```
for i := 1 to n
   for j := 1 to n
      if a[i] > a[j] then exchange(a[i],a[j])
```

or, after a slight modification, to

```
for i := 1 to n
   for j := i+1 to n
      if a[i] > a[j] then exchange(a[i],a[j])
```

$$\forall\, i\ j \mid i<j \Rightarrow a[i] \leq a[j]$$

An alternative definition might be given recursively. A sequence of cardinality one is ordered by definition. In formula:

$$\text{cardinality(A)=1} \Rightarrow \text{ordered(A)}.$$

If cardinality (A)>1, let us suppose that a subsequence B of A of cardinality n-1, is ordered. In order to infer that the sequence A is ordered one has to distinguish three cases depending on the relative position of the left over element which can be less than any element of the subsequence B or greater than or intermediate. In the first two cases we have that a recursive definition of ordered (A) might look like

(2) $$\text{ordered(A)} \Leftrightarrow \text{ordered(B) and } B \leq C \text{ and cardinality(C)=1 and A=B cat C}$$

(3) $$\text{ordered(A)} \Leftrightarrow \text{ordered(B) and } C \leq B \text{ and cardinality(C)=1 and A=C cat B}$$

where $B \leq C$ means that every element of B is less or equal than every element of C and cat is the operation that applied to two ordered sequences B and C such that $B \leq C$ gives a sequence A that happens to be ordered. In the third case we define a merge operation as an operation that, applied to two ordered sequences, gives a third sequence which happens to be ordered. In formulas:

(4) $$\text{ordered}(A) \Leftrightarrow \text{ordered}(B) \text{ and cardinality}(C)=1 \text{ and } A=\text{merge}(B,C)$$

A similar analysis shows that the splitting of A into B and C can be more general or that the condition cardinality(C)=1 can be released replacing it with ordered(C). We have

(5) $$\text{ordered}(A) \Leftrightarrow \text{ordered}(B) \text{ and ordered}(C) \text{ and } B \leq C \text{ and } A=B \text{ cat } C$$

(6) $$\text{ordered}(A) \Leftrightarrow \text{ordered}(B) \text{ and ordered}(C) \text{ and } A=\text{merge}(B,C)$$

The formulas (2), (3) and (4) are in essence the guidelines for the sorting methods known as bubble-sort by minimum or by maximum or by insertion. Formulas (5) and (6) are the base for quicksort and mergesort. Even formula (1) can be used as a guideline for some methods, in particular the method comparing each element with all the others, counting the elements which are less than it and using such counters as indexes for their relative positions.

Obviously, one can easily show that the six formulas are equivalent. For instance formula 1 is equivalent to

$$\forall p.q \mid p<q\leq n-1 \quad a[p]\leq a[q] \quad \text{and} \quad \forall r \mid r\leqslant n-1 \quad a[r]\leqslant a[n]$$

i.e. to formula (2). One could certainly start from one formula, say (1) taken as the unique problem specification, and try to derive all the sorting programs from it (see Darlington |76|) but that process seems to suffer from an excess of formality, useful only if used by some automatic derivation system.

Alternatively, one could start from one specification, derive a specific program and then perform a set of program transformations to derive from it all the other sorting methods. In fact programs can be derived in a variety of ways according to different styles of derivations. We think that one should try to keep the program derivations at the highest level of abstraction for the greatest number of steps. This tactic implies using a variety of equivalent formulations of the problem to be solved or in other words implies to start the optimization job right from the beginning at the specification level, as we have done for the sorting methods.

5. DERIVATIONS OF PROGRAMS

We shall illustrate and comment on two single derivations of programs. These will be examples of what we intend by rigorous but informal derivations. The first is the trivial example of the integer square root program, the second is the minimal spanning tree of a graph. Since the latter is a little longer it will be given in the Appendix. The derivations will be given as successive photographs of the program plus comment describing the modifications or transformations.
Of course the first form of the square root program is the following program, directly obtained from the specification:

```
i  := 0;
while not $i^2 \leq x < (i+1)^2$ do
   i := i+1
 $\{ i^2 \leq x < (i+1)^2 \}$
```

The first optimization consists in noting that the while test is redundant because if $i^2 \leq x$ before the loop then, after the assignment i:=i+1, one still has $i^2 \leq x$. In other words $i^2 \leq x$ is a program invariant.

```
i  := 0;
 $\{ i^2 \leq x \}$
while not $x < (i+1)^2$ do
   i := i + 1
$\{ i^2 \leq x < (i+1)^2 \}$
```

Since $(i+1)^2 = i^2 + 2i+1$ we need not to compute each time $(i+1)^2$ anew, but we may profit of the previous value stored into a variable, called i-plus-one-square, adding each time 2i+1 to it. Again, instead of computing 2i+1 anew each time, we introduce a second variable two-i-plus-one. We have:

```
i := 0;
two-i-plus-one := 1;
i-plusone-square := 1;
{ $i^2 \leq x$ and two-i-plus-one = 2i + 1
      and i-plus-one-square = $(i + 1)^2$ }
while not x $\leqslant$ i-plus-one-square do
  [ i := i + 1;
    two-i-plus-one := two-i-plus-one + 2;
    i-plus-one-square := i-plus-one-square + two-i-plus-one
{ $i^2 \leqslant x < (i+1)^2$ }
```

which is our final square-root program. It is more natural to communicate the "meaning" of the square-root program by saying that it is a linear axhaustive search optimized by simplification of the loop-test and by introducing two state variables (to use the relation $(i+1)^2 = i^2+2i+1$), than by any other way, which, using correctness proofs or hand simulation, tries to express a meaning in terms of the final program. In other words, the derivation process is a practical way to transmit the meaning of this program to a person: programmer or developer. We might even reverse the usual attitude towards program correctness. No longer do we care whether the written presentation of the previous algorithm contains bugs or typing error if the given derivation method makes us feel secure and confident to be able to reproduce the derivation process with the requested degree of precision. Absolute precision is needed in fact only for those programs which we do not know how to derive. Absolute precision is of fundamental importance only if we ignore all about an algorithm; we only know that "it works and therefore we must not touch it". For the same reasons absolute and detailed program correctness is fundamental for machine execution. A treatment analogous to the square root derivation can be easily given for other elementary programs.

Incidentaly if anyone tried to remember final programs and correctness proofs of the simple examples reported in the classical text-book of Manna |76| one would probably fail 90% of the time while it is just too trivial to derive them through a series of derivation steps.

Now a few more comments. First, if the programmer of the previous algorithm does not see immediately that $i^2 \leqslant x$ is a cycle invariant, he will come out with a little less optimized program having one more program variable (i-square) and a slightly longer cycle test, but still a reliable program properly annotated with a correctness proof. In other words, a tree of derivations is a stable process whose intermediate results are still of pratical value; it is not an all-or-nothing process.

Second, the previous derivation steps not only easily allow you to derive the square root-program, but immediately lead you to generalyze it into a similar program, say, for the cubic root, since $(i + 1)^3 = i^3 + 3i^2 + 3i + 1$, and more generally, induce you to exploit recurrence relations in order to optimize programs defined by specifications like: $\forall x \quad y \quad P(x,y)$.

Third, a final comment on a taxonomy of programs. Derivations of programs naturally induce a relationship of programs that are derived through the same derivation pattern and it will be natural and interesting to study programs from this point of view. But then, similarity of derivations may be quite unexpected. For instance, the sum of two symbolic polinomials, represented say by linked lists, and the program for merging two ordered sets do share the same derivation.

Can we delimit or describe the concept of derivation of a program? Certainly, such a concept will include program derivations by step-wise refinements, by transformations, by modifications and by applications of program schemata |Gerhart 76|. Step-wise refinements are well known and will not be discussed here. Transformations usually denote a program modification that leaves invariant the computed result. Transformations techniques have been widely studied: they include elimination of recursion in certain cases and generic source optimizazions. A class of transformations, better called program synthesis, deserves to be mentioned: it studies the transformation of static specifications into recursive definitions (by folding and unfolding). Program modification slightly changes a program keeping invariant only some objectives in order to solve a slightly different problem. Of course once a derivation pattern has been secured one should try to generalize it into a derivation of a schematic program or of an abstract program and, for the moment, leave to the ingenuity of the designer which schema or schematic pattern to apply in each concrete case. However, more than on abstractly studying classes of program derivations our emphasis in this research program is on concretely applying derivations of programs to specific cases as an everyday working tool of a teacher when he/she tries to "present" a program to his student or of a designer when he/she tries to document an implementation.

6. A TOOL FOR DERIVATIONS OF PROGRAMS

Programs are derived from specifications in steps and the initial phases of the derivation require the ability to express abstract algorithms. One example of abstract algorithm is the minimal spanning tree algorithm of the appendix; abstract algorithms are being currently used in the literature (see books by Aho, Hopcroft and Ullman or S. Baase). Not surprisingly abstract algorithms are expressed in an informal language, but nevertheless they can be considered to be unambiguos and rigorous (to adopt "our" terminology). Such a program design language is a sort of documentation language whose purpose is to convey ideas and not to instruct machines.

An important objective to be obtained is the derivation of the initial program design into the final implementation. The derivation must allow an easy link of each piece of program code to the design ideas that originated it in the first place. A research program on tools for program derivation has given origin to the system DUAL, described in |Petrone 82,83|.

Briefly, DUAL is an intelligent editor/incremental compiler designed to cope with the evolutionary nature of the design process as embodied by the techniques of program transformations and stepwise refinements. DUAL takes special advantage of the screen-keyboard interface to provide a natural access to the tree-structured design data base where design informations and program texts are strictly correlated.

The various steps of the design process are "photographed" by DUAL and the "movie" showing the derivation phases can be dinamically "replayed" on the screen at the visitor's (designer or maintenance-developer) will. A clear distinction between design, implementation and documentation no longer exists in DUAL. The description of a design is a description of the implementation at a higher level of abstraction and is expressed in an informal design language which, by successive modifications, will derive into the program text.

REFERENCES

Baase S. |1978| Computer Algorithms: introduction to Design and Analysis. Addison-Wesley.

Balzer R. |1981| Transformational Implementation: an example. IEEE Trans. on Soft. Eng., Vol SE-7, 3-14.

Bauer F.L. |1976| Programming as an evolutionary process. Proc. 2-nd Int. Conf. on Soft. Eng., San Francisco, 223-234.

Bauer F.L. and Wössner H. |1982| Algorithmic Language and Program Development. Springer Verlag.

Burstall R.M. and Darlington J.A. |1977| A transformation system for developing recursive programs. J. ACM 24, 44-67.

Darlington J.A. |1976| A synthesis of several sorting algorithms. Res. Rep. N.23, Dpt. of A.I., Un. of Edinburgh.

De Millo R.A., Lipton R.J. and Perlis A.J. |1979| Social processes and proofs of theorems and programs. Comm.s ACM, Vol 22-5, 271-280.

Dijkstra E.W. |1972| Notes on structured programming in Structured Programming, Academic Press.

Gerhart S.L. and Yelowitz L. |1976| Control Structure abstractions of the Backtracking Programming Technique. IEEE Trans. on Soft Eng.

Manna Z. |1976| Mathematical Theory of Computation. MC Graw-Hill.

Manna Z. and Waldinger R. |1979| Synthesis: dreams = programs. IEEE Trans. Soft. Eng., SE-5.

Naur P. |1982| Formalization in program development. Bit, 22, 437-453.

Petrone L. et alii |1982| DUAL: an interactive tool for developing documented programs by step-wise refinements. Proc. 6-th Int. Conf. Soft. Eng., Tokyo, 350-357.

Petrone L. et alii |1983| Program development and documentation by step-wise transformations: an interactive tool. Proc. Int. Comp. Symp. Nürnberg.

Scherlis W.L. and Scott D.S. |1983| First steps towards inferential programming. IFIP Congress, 199-212.

Waters R.C. |1979| A Method for analyzing loop programs. IEEE Trans. Soft. Eng., 5.

Wirth N. |1971| Program development by step-wise refinements. Comm. ACM, 14, 221-227.

APPENDIX

An example of derivation of programs: minimal spanning tree of a graph. (We reformulate in terms of derivations an example taken from the book of S. Baase).

Given an undirected graph G=(V,E,W) where V, E, W are the sets of nodes, edges and associated weights, let us represent a spanning tree T of G as a subset of E.

The first important step of the derivation is to reformulate a theorem in a recursive form so that the algorithm is but a trivial implementation of the recursion.

Let us first introduce the following notations. Given a graph G=(V,E,W), we want to consider the "complete" subgraphs $G_i=(V_i, E_i, W_i)$ of G which are unguely obtained in correspondence with subsets V_i of V by assuming that G is made of all the edges E_i of E whose vertices belong both to V_i. If $G_i=(V_i, E_i, W_i)$ is a subgraph of G then <u>adjacent(V_i)</u> is the set of all vertices of G which are not in V_i but are adjacent to same vertex of V_i and <u>incident(V_i)</u> is the set of all edges of E which are not in E_i but which are incident to vertices of G_i. Let P and Q denote the predicates "V_2 = adjacent(V_1)" and "E_2 = incident(E_1)" respectively.

Definition. The subgraph consisting of any vertex x of G=(V,E,W) is minimal. If a subgraph G_i =(V_i) is minimal and xy is the edge of minimal weight belonging to incident(V_i) then G_{i+1}= (V_i u { y}) is minimal. Note that if G is connected G is trivially a minimal subgraph of itself. Now we can state the recursion theorem:

Theorem. The subgraph G_1 consisting of any vertex of G=(V,E,W) is minimal and its minimal spanning tree is the empty set. If G_i=(V_i) is a minimal subgraph of G, and T is its MST, then if we denote with xy the edge of minimal weight belonging to incident(V_i) we have that G_{i+1} =(V_i u{x}) is minimal and T_{i+1} =T_i u {xy} is the MST of G_{1+1}

A first high level description of the algorithm is a straighforward transliteration of the recursion theorem and is the application of the well-known schema of building a program "around" a cycle invariant

```
    $V_1$ <- any vertex of G
    $E_1$ <- empty set
    $V_3$ <- V -  x

{assert: the subgraph denoted by the set V is minimal}
{assert: E is a MST for the subgraph denoted by V}

"make the predicate P true"
"make the predicate Q true"

{assert P,assert Q}

while $E_2$ not empty  do
    find an edge e in $E_2$ of minimum weight;
    set x to be the vertex in $V_2$ incident with e
    $V_1$ <- $V_1$ u {x};
    $V_2$ <- $V_2$ - {x};

    $E_1$ <- $E_1$ u {e};
    $E_2$ <- $E_2$ - {e};

   {assert:  the subgraph $G_i$, denoted by $V_1$, is minimal}
   {assert: $E_1$ is a MST for the subgraph denoted by $V_1$ }

    "restore predicates P";           "restore predicates Q"
```

```
  {assert P, assert Q}
 end
```

Note that when E_2 becomes empty, E_1 has grown to include all vertices belonging to one connected component of G.
The operations restoring predicates P and Q are simple, they are omitted. A first optimization is to make the set E_2 containing one edge (the one with minimal weight) for each vertex in V_2. This effects the definition of Q and the operation restoring that predicate which now becomes

```
     for each y in $V_2$ adjacent to x do
         if weight(xy) $<$ weight(the edge e in $E_2$ incident with y)
           then $E_2$ <- $E_2$ - {e} u {xy}
```

The operation restoring predicate P becomes

```
    for each y in $V_3$ adjacent to x do
       $V_2$  - $V_2$ u {y};
       $V_3$  - $V_3$ - {y};
       $E_2$  - $E_2$ u {xy}
```

A second optimization is a space/time optimization. The operations restoring P and Q occur twice in the program. They can be moved from outside into the loop, their order can be inverted and the final program becomes similar to the abstract algorithm reported in Baase which is an unstructured program.

Incidentally, unstructured programs could be allowed in a derivation tree. After all, the final program texts of derivations are no longer used to convey the meaning of a design. If we revisit all the phases of this derivation we see that are simple and natural. The only points where some ingenuity is required are the initial recursion theorem and the restrictions of the sets where the search of minimum cost edge must be performed.