Part One

Pragmatic Versus Structured

Computer Programming

A Theory Beyond Criticism (1975-1983)

Chapter 1

Introduction

1.1 Background

It is not hard to understand why a new science such as computer programming would adopt the methods and goals of mathematics which is highly successful and whose roots date back to antiquity. The use of computers in solving mathematical problems and the mathematical training of many senior computer scientists only serve to increase this likelihood. Since formal mathematics in the guise of metamathematics and logic has dominated mathematical thought throughout this century, it is also not surprising that computer science would adopt a formal methodology. This methodology is called structured programming and has until recently been almost universally accepted.

This thesis maintains that structured programming is incorrect in the sense of being both problem ridden and detrimental to the development of new programs. It argues that computer programming is a pragmatic activity in which progress requires problem and individual programmer specific methods (tricks). Methods are indicated not only by the specific nature of the problem or subproblem being solved, but also by the background knowledge in which the problem is embedded and the aptitudes and background of the individual programmer.

The argument is that while mathematics is well suited to solving problems involving abstraction and generalization, computers are well suited to problems which can be reformulated or tailored to allow the use of computational tricks. Of course, the programming solutions to mathematical problems will still involve normal mathematical methods, and where formal algorithm analysis is possible, the analysis may involve formal proofs. The point is that the computer solution to large and complicated problems is not formal and cookbook like in the structured programming sense.

1.2 The Nature of Structured Programming

Structured programming offers methods for all three phases of the computer programming process: design, verification, and implementation. According to structured programming, programs (or algorithms) are designed by means of refinement which is a top down design or improvement process in which an abstract conception is systematically transformed into an actual program. Verification of previously written programs is guaranteed by formal mathematical proofs. Although, it is sometimes claimed the proof and the implementation must go hand in hand. Structured programming guides the implementation process by providing stylistic guidelines and programming language features which are claimed to aid the refinement and verification processes, and by forbidding other features which are claimed to be either mathematically unsound or not systematic.

1.3 Argument Overview

This thesis contains various types of scientific and philosophical evidence against two of the three facets of structured programming. It also presents by means of examples an alternative pragmatic and problem specific approach to computer programming which is free of cookbook like methods.

It is often difficult to find evidence against abstract methods since they almost always have a psychological component and are almost never specified explicitly. In the case of structured programming, it has been even more difficult since, when this thesis was begun in the middle 1970s, structured programming was universally accepted and commonly held to be beyond criticism. In addition, structured programming's area of applicability is not some objective physical measurement but rather the product of human problem solving. In order to argue against such a theory, it is first necessary to establish that its correct application still leads to mistakes or difficulties.

Chapter two establishes that at least the refinement part of structured programming has problems. An application of refinement to a simple programming problem in which a founder of structured programming ran into difficulties is discussed. Since the difficulties occurred in a graduate level textbook, in which the author himself claimed to be illustrating the nature of the principle, the failure must be caused by the unsoundness of the technique.

Another approach to arguing against a method is by convincing philosophical arguments. Chapter three contains a dialogue in which various computer scientists attempted to defend structured programming by explaining why the material in chapter two should not be published in the scientific journal titled *Communications of the ACM*. This gives me an opportunity both to rebut their arguments and to explain my pragmatic alternative. Such a verbatim dialogue is advantageous since it prevents any participant from claiming that the disputed claim is either not part of structured programming or not that person's viewpoint. It also offers a vehicle for rebuttals and counter-rebuttals.

Chapter three attempts to show that the followers of structured programming do not see it as open to possible refutation and are not willing to specify conditions under which they would abandon it. This argument is strengthened by the seeming unwillingness of *CACM* editors to base editorial decisions solely on technical content.

Chapter four argues against verification by means of program correctness proofs. It provides a list of programs and problems which should be provable if verification is to retain any promise.

Finally, it is not sufficient to simply find flaws in a widely held theory. Some alternative must be presented. The problem in presenting alternative methods in the area of human problem solving is that there is a tendency to interpret any concrete approach as a new and improved formal method which simply has replaced the old flawed one. The real alternative is no universal, problem independent method, but rather that any particular problem must be solved using an approach dictated by the nature of the problem. The alternative pragmatic approach is illustrated in chapter two by showing an improved solution to the Dutch national flag problem, and in chapter five by presenting the solution to an economically important subproblem of the integrated circuit gate array placement problem. An attempt has been made in both cases to explain the relevant problem specific knowledge. Chapter five also contains a brief explanation of why the approach used to place CMOS (a type of integrated circuit manufacturing technology) gate arrays does not work particularly well for ECL array placements.

The reader who expects a tightly organized and linear argument will be disappointed. Each main chapter was written to present the entire argument and to stand alone. The organization is chronological with each chapter written, at least in part, to answer objections raised with respect to previous ones. The reader only interested in new algorithms should read chapters two and five while the reader only interested in philosophy should read chapters three and four.

1.4 Omission of Stylistic Issues

The third phase of structured programming which involves programming style and language choice is not discussed here. The value of structured programming based languages is currently being decided in the programming laboratory (work place). Practicing programmers and their institutions are carrying out the test by choosing between languages such as Pascal or ADA which embody and enforce the stylistic tenets of structured programming and C which does not. It seems that C with its pragmatic features is prevailing, but it may be too early to tell. For an interesting comment on this question see appendix A.

1.5 Publication History

Most of the material in this thesis was written to appear somewhere else and has therefore been left as close to the original form as possible. Changes have been limited to the correction of obvious typographical errors and to consistent formatting. In the case of cited references, if a more accessible or improved version has become available, that reference is used.

Chapter two was submitted to the *CACM* and appeared as Zilog Corporation technical report number five. A new postscript discussing recent work on the Dutch national flag problem has been included. Chapter three is a verbatim transcript of an exchange with various ACM officials over the publication of chapter two. Names have been omitted to emphasize the official nature of the exchange. Chapter four appeared in the ACM SIGSOFT newsletter. It also contains a postscript discussing related current papers. Chapter five appeared in the proceedings of the 1983 IEEE Conference on Computer Design VLSI in Computers (ICCD), but the postscript discussing ECL gate array placement did not appear in those proceedings.

Chapter 2

A Failure of Structured Programming

2.1 Introduction

Refinement lies at the heart of structured programming. Algorithms are designed using stepwise refinement,[1] data structures are chosen using static refinement,[2] and algorithm efficiency is improved using optimizing refinement.[3] Refinement is a top down program design or improvement process in which an abstract conception is systematically transformed into an actual program.[4] This paper examines a failure of optimizing refinement. It indirectly casts doubt on structured programming due to the important role refinement plays in structured programming and the similarity of all types of refinement.

The unsuccessful refinement occurred in Professor Dijkstra's attempt to improve a solution to the Dutch national flag problem.[5] It involved the transformation of a short unrefined program into a longer but no more efficient one. Situations were identified in which the unrefined program performed extra computation, but in handling those situations, decisions were made which necessitated increased computation in other cases. The overly rigid nature of refinement seems to have prevented the discovery of a truly more efficient solution. Such a solution, discovered by considering a simplified version of the problem, is presented.

2.2 The Problem

The problem involves exchanging colored pebbles within a row of buckets to satisfy a given condition. Input is a row of N buckets each of which is filled with a colored pebble chosen from the three colors: red, white, or blue. The pebbles must be rearranged so all red pebbles are contiguous and to the left of all blue and white pebbles. The white pebbles must also be contiguous and lie to the left of all blue pebbles (this happens to be the order of colors on the Dutch national flag). The solution must be written exclusively in terms of two primitives:

buck(i): This function returns the color of the pebble in bucket i.

swap(i,j): This procedure exchanges the pebble in bucket i with the one in bucket j.

Dijkstra specifies a number of additional constraints.[6] The buck(i) operation may be used on a given pebble only once since the operation is assumed to be so time consuming that any other solution would be unacceptably slow. The normal two-step approach of storing the color of each pebble in an array and then outputting a sequence of swap operation, possibly using some kind of Quicksort,[7] is ruled out. Only simple variables with a range not much larger than N are allowed. Finally, given programs with the "the same degree of complication, the one that needs (on the average) the fewest swaps is to be preferred".[8]

This problem is not very interesting in itself. None of the solutions are particularly deep, and the problem has been so constrained that it is hard to imagine it ever arising in real programming. Yet, it is interesting because of the light it sheds on structured programming. It was published to investigate "a more refined solution"[9] which Professor Dijkstra believed to be "elegant".[10] It is complicated enough to provide a real test for refinement, yet simple enough to be easily described and results in procedures short enough to be shown in their entirety. Finally, the wide acceptance of structured programming seems mainly due to philosophical arguments for its basic tenets[11] rather than to any careful examination of the results of its

---

1. Wirth[1971b], p. 221.
2. Dijkstra[1972a], p. 59.
3. *Ibid*., p. 43.
4. *Ibid*., p. 41.
5. Dijkstra[1976], pp. 111-116.
6. *Ibid*., p. 112.
7. Cf. Hoare[1962], 10-15.
8. Dijkstra[1976], p. 112.
9. *Ibid*., p. 114.
10. *Ibid*., p. 117.
11. Dijkstra[1972b], 859-865.

application. This problem allows such an examination to be carried out.

2.3 The First Solution

The unrefined solution requires the observation that there are really four kinds of buckets: buckets known to contain pebbles of one of three colors and uninspected buckets. The buckets can be divided into four regions with the unexamined region somewhere in the middle. The unexamined region begins as the entire set of buckets and gradually disappears. Notice the boundary variables always point to the next place to put a pebble of the appropriate color. During execution of the program the row of buckets might look like:

```
-----------------------------------------------------------
|examined red |unexamined |examined white |examined blue |
-----------------------------------------------------------
1             r         w               b              N
```

where the following conditions are satisfied:

```
1 <= k <  r    the kth bucket contains a red pebble

r <= k <= w    the kth bucket contains a pebble whose
               color is unknown

w <  k <= b    the kth bucket contains a white pebble
               the kth bucket contains a blue pebble
```

The program examines the wth bucket and puts its pebble into the appropriate region. It requires on the average 2/3 N swaps. Professor Dijkstra gives the solution shown in figure 1.[12]

```
procedure dutch_national_flag;
  var b,r,w: integer;
begin
  r := 1; b := N; w := N;
  while w <= r do
    case buck(w) of
      red:  begin swap(r,w); r : = r + 1; end;
      white: w := w - 1;
      blue:
        begin swap(w,b); w := w - 1; b := b - 1; end;
    end
end;
```

Figure 1. Simple DNF Program

This program is clearly simple but may do extra swapping. For example, if all pebbles are red, it will execute N swaps when none are required. It would also be advantageous to place two pebbles with one swap.

2.4 Dijkstra's Refined Solution

Dijkstra discovered his solution by refining the first solution. He claims to have used the following thought process:[13]

---

12. Dijkstra (*Ibid*., p. 114) published this program in his own non-deterministic programming language. I have used a slightly modified version of Pascal (Wirth[1971a], pp. 35-63) which I think makes the paper easier to understand.

    This procedure assumes various objects have already been defined such as the type color and the value of N.

    As Professor Dijkstra observed, it is important to notice that the solution which examines the rth bucket and moves left to right requires more swaps since blue pebbles require two swaps.

13. *Ibid*., p. 114-116.

1. He notices that unnecessary swaps of red pebbles can occur.
2. He modifies the algorithm to move the boundary of known reds (r) to the right as much as possible without swapping.
3. He notices that the red boundary may have reached the boundary of known whites (r = w) and decides to handle that case immediately with a case statement similar to the one in his first solution.
4. He notices that when r < w and a non-red pebble is encountered (at r), some other pebble must be inspected or the algorithm reduces to the inefficient one which just moves from left to right (see footnote 12. above).
5. He chooses to inspect the one in bucket w.
6. He sees that if the pebble in the wth bucket is white, the algorithm should attempt to move the known white boundary (w) until finding a non-white pebble or until w = r + 1.
7. The two inspected pebbles which must be in different buckets (from number 3 above) need to be handled. There are two possible colors in bucket r (white or blue) and all three are possible in bucket w. There are then six cases to handle, but the two cases involving a white pebble in bucket w only occur when w = r + 1.
8. He decides to place the pebble originally in bucket w first and make sure the pebble which was in bucket r ends up in the new wth bucket.

See figure 2 for a straight translation of Dijkstra's solution to Pascal.[14] Dijkstra's refined solution requires on the order of 2/3 N swaps which is exactly the number required by his unrefined solution and 1/9 N swaps more than my solution given an even distribution of colors. No matter how large N becomes, the improvement of Dijkstra's refined solution over his first solution is always less than 1/2 of one swap.[15] Dijkstra's refined solution handles the unlikely case of long strings of red or white pebbles more efficiently than the unrefined algorithm, and sometimes correctly places two pebbles with one swap, but nullifies this improvement by swapping uninspected pebbles which must later be swapped back to their proper region.

Dijkstra's refinement process allows him to notice that more than one pebble must be placed at a time, but he lacks any guiding principle for doing the placing. He errs at step 8 above. He should handle pairs of buckets but instead decides to first do something with bucket w and then do something with bucket r. As long as bucket w contains a red or white pebble his approach is good, but if the pebble in bucket w is blue, the refined algorithm places the blue pebble in its proper place, but then moves w one to the left and swaps whatever is in the new bucket w (it has not been inspected) with the pebble in bucket r. If the pebble in the new uninspected bucket w is blue or white, his algorithm makes two extra swaps. If it is white, it gets swapped into bucket r and swapped back the next time through. If it's blue, it goes to r then back to w and finally to b, but one swap would have been needed in any case.

Not only is the refined algorithm inefficient and complicated, but also the program written to express it is longer than necessary. The code for the first case statement which is executed only when w = r after skipping over red pebbles (see figure 2) is almost exactly duplicated in the last case statement of the program. The first case statement could be completely eliminated by moving it outside its guarded block and by adding code to handle the colr = red case. This would considerably shorten the program without changing the algorithm. It is impossible to determine whether this program's lengthly style is related to the use of refinement.

2.5 Another Solution

Dijkstra proceeded by refinement. I proceeded by trying to understand the problem better. This solution was discovered by considering a simpler problem with pebbles of only two colors (red and white) and extending that solution to handle three colors.[16] It repeats the following three steps until all the pebbles

14. Professor Dijkstra read an earlier version of this paper and pointed out that I had misinterpreted his description of how to compose the refined program. I am grateful to him for sending me the version which appears here.
15. McMaster[1979], 842-846.
16. The idea to consider the two color case first arose during a discussion with Dana Angluin, Larry, Ruzzo, and Amiram Yehudai.

```
procedure dutch_national_flag;
  var b,r,w: integer; colr,colw: color;
begin
  r := 1; b := N; w := N;
  while r <= w do
    begin
      colr := buck(r);
      while (colr = red) and (r < w) do
        begin r := r + 1; colr := buck(r) end;
      if r = w then
        case colr of
          red:   r := r + 1;
          white: w := w - 1;
          blue:
            begin swap(w, b); w := w - 1; b := b - 1; end
        end;
      if r < w then
        begin
          colw := buck(w);
          while (colw = white) and (r + 1 < w) do
            begin w := w - 1; colw := buck(w); end;
          case colw of
            red:   begin swap(r, w); r := r + 1; end;
            white: w := w - 1;
            blue:  begin
              swap(w, b); w : = w - 1; b := b - 1;
              swap(r, w);
            end
          end;
          case colr of
            white: w : = w - 1;
            blue:
              begin swap(w, b); w := w - 1; b := b - 1 end
          end
        end
    end
end;
```

Figure 2. Dijkstra's Refined Program

have been inspected (this can occur during steps 1 or 2):

1. Start from the left and inspect pebbles until a non-red one is found.
2. Start from the right and inspect pebbles until a non-white one is found.
3. Swap the two pebbles and continue with step 1 from where step 1 and step 2 left off.

This algorithm is optimal. Consider the sum of the distance of red pebbles from the left end. Each swap reduces this sum by the maximum possible amount.

This algorithm is easily modified to handle blue pebbles. If a blue pebble is found while looking for a non-white one, the blue one can be swapped with the white one at the blue white boundary and step 2 can be repeated from that point. This insures that when the algorithm gets to step 3, the non-white pebble is red. There are then 2 cases:

[white, red] simply swap as in the 2 color algorithm

[blue, red] two swaps will place both pebbles in their correct regions (swap(r,w) and swap(w,b))

Notice that a red or white pebble is already in its proper place. This solution requires on the order of 5/9 N swaps.[17] See figure 3 for a possible Pascal procedure.[18]

```
procedure dutch_national_flag;
  var b,r,w: integer; colr,colw: color;
begin
  r := 1; b := N; w := N;
  while w >= r do
    begin
      for r := r to w do
        begin
          colr := buck(r);
          if colr <= red then goto skip_whites
        end;
      return;
skip_whites:
      for w := w downto r + 1 do
        begin
          colw := buck(w);
          if colw = blue then
            begin swap(w,b); b := b - 1 end
          else if colw = red then goto make_swaps
        end;
      if colr = blue then swap(r,b);
      return;
make_swaps:
      swap(r,w);
      if colr = blue then
        begin swap(w,b); b := b - 1; end;
      r := r + 1; w := w - 1;
    end;
end;
```

Figure 3. More Efficient Program

2.6 Conclusion

In my opinion, this failure is no special case. It is not the one exception to a workable design method. The Dutch national flag problem was created by one of the originators of structured programming to illustrate refinement.[19] Therefore, it plays a paradigmatic role in the elucidation of refinement and even one such failure casts doubt on the whole method.

If one can make a useless refinement in this simple and artificially constrained problem, it is hard to imagine how refinement could be successfully used in complex situations. The suggestion that one try to improve some idea is certainly valuable, and in many cases inexperienced programmers need a concrete starting point, but refinement leads to difficulties because it is a far too restricted approach for general use. Structured programming has taken a piece of good advice and turned it into an inflexible design rule. In this example, it caused a good programmer to only think along a few specific lines instead of searching for the new insight which would have allowed the discovery of a more efficient solution.

The point of this paper has been eloquently expressed by Feyerabend in a book on rationalism and science.

---

17. *Ibid*.
18. The reader may have noticed I have taken some liberty with Pascal as it is defined in the Pascal report (Wirth[1971a], pp. 35-63). n addition to a few cosmetic changes, I have added identifier labels and return statements, and assume for loop index variables are defined outside their loop and upon termination have a value one past the final value. I find these changes important for my style of programming.
19. Dijkstra[1976], p. 114.

> The idea that science can, and should, be run according to fixed and universal rules, is both **unrealistic** and **pernicious**. It is unrealistic for it takes too simple a view of the talents of man and the circumstances which encourage, or cause, their development. And it is pernicious, for the attempt to enforce the rules is bound to increase our professional qualifications at the expense of our humanity. In addition, the idea is **determental to Science**, for it neglects the complex physical and historical conditions which influence scientific change. It makes our science less adaptable and more dogmatic: every methodological rule is associated with cosmological assumptions, so that using the rule we take it for granted that the assumptions are correct.[20]

2.7 Postscript

2.7.1 DNF Algorithm Efficiencies

This paper resulted in a number of publications one of which includes a possibly improved algorithm. Professor Bitner[21] discovered and analyzed an algorithm which guesses that pebbles are exactly evenly distributed (1/3N of each color) and immediately starts placing each pebble into its expected region. Under the condition of an even distribution of colors, it clearly produces a more efficient algorithm (10/27N versus 5/9N). The possible problem with this approach is that the solution produces an average case improvement for one particular distribution while not improving the worst case (still 2/3N) and results in a program three times as long.[22] I actually briefly (too briefly?) considered the guess approach but no practical programmer would bet on such regular data. A real program must somehow balance the conflicting requirements of average case performance, worst case performance, measured performance, and finally program size.

Bitner also discovered a method to allow the simulation of an array for storing the color of each pebble using only a small number of extra memory locations (called scouts). This effectively allows more than two pebbles to be inspected at once. He showed that the guess algorithm using scouts asymptotically approaches the average case optimum of 1/3N.
The average case results are:

1. asymptotic guess algorithm with scouts - 1/3N.
2. guess algorithm without scouts - 10/27N.
3. two pass binary algorithm which violates one of Dijkstra's constraints (see 3.12) - 7/18N.
4. my extended binary algorithm - 5/9N.
5. Simple 12 line algorithm - 2/3N.

Two other analyses of Professor Dijkstra's refined algorithm appeared as preliminary research reports.[23] More recently Professor Dijkstra discussed the DNF problem from a different perspective in his book of selected writing.[24]

2.7.2 Defenses of Refinement

Even though this chapter was never published, Professor Floyd seems to attempt to answer the objections it raises in the part of his turing lecture that defends refinement as a correct paradigm.[25] He defends refinement and rule like knowledge while agreeing that structured programming may be too narrow.[26] He also thinks *A Discipline of Programming* is not one the better expositions of structured programming (see his 1977 review of that book).[27] He seems to be defending the concept of rule like knowledge and therefore argues for the possibility of artificial intelligence (but see Lighthill's skeptical report on AI).[28]

---

20. Feyerabend[1975], p. 295.
21. Bitner[1982], pp. 243-262.
22. *Ibid*., p. 259-261.
23. Cf. Jonassen[1977] and Gotshalks[1978].
24. Dijkstra[1982], pp ??.
25. Floyd[1979], pp. 455-460.
26. *Ibid*., p. 456.
27. Floyd[1977], p. 785.

Chapter 3

A Dialogue on Structured Programming

3.1 First Version Acknowledgement Letter

From: Editor 1

To: Author

Date: January 18, 1977

Since [Berkeley Professor] was one of those who provided you with advice on this paper, I will accept it and process it. Please tell [Berkeley Professor] that I am doing this, as it is clearly in [Berkeley Professor's] area of our joint editorship! If [Berkeley Professor] wishes to take over, I will send [Berkeley Professor] the file immediately.

The paper illustrates a point on which I have felt much as you do for some time, and I hope that does not introduce too much bias *in favor* of it! One matter of definition I have never understood is this: If one views the stepwise refinement abstraction process as the creation of an "abstract tree," then clearly programming by refinement means that one successively adds material to the frontier of the partially formed tree. This is clearly highly non-"context free" in the sense that one uses insights gained from the parts of the tree already constructed to add nodes to the frontier. (An example is Dijkstra's account of the 8 queens problem, in which the evolving program established the need for data structures necessary to hold information about the diagonals). Nevertheless, the rules of the game are still clear.

Now comes the fly in the ointment. Some people (e. g. Dan Bobrow) say that it is fair to first (1) judiciously select high-level data structures, (2) interface routines for these data structures, and (3) complex procedures.

The abstraction tree is shallower than before, because nonprimitive data operations and procedure calls lie at its frontier. In short, the tree may be put together so that *preconstructed subtrees* are attached as leaf nodes.

What puzzles me is whether (1) this is really programming by refinement, and (2) whether *any style whatever* cannot be dressed up to appear to be programming by refinement, i.e., whether refinement has much meaning at all.

Of course, one can hedge one's bets, and hypothesize that *most of the process of program creation - the least creative part* - is authentically top-down, and some bottom-up aspects - frequently the most creative - may be systematized by inventing complex data structures and/or procedures, and then seeing how they work into the top down process, changing both if necessary as one goes along. Maybe this weakened form of programming by refinement describes what people really do, but if so, it certainly does not have anything like the simplicity claimed by its proponents.

I would be glad to hear your reactions. In the meantime, you will be happy to hear that I am referencing your paper (along with 40 others) in a paper on strong loops and selectors ("generators" in ALPHARD).

3.2 Dijkstra's Response to First Version

From: Professor Dijkstra

To: Author

Date: January 24, 1977

for the paper that you mailed on Jan. 16, 1977, I am --alas-- forced to conclude that you have poor helpers and supervisors. On pg. 116 I have written "I leave the final composition of our second program to the reader...", and apparently no one in your environment has observed, that in this case the reader --you to

28. Lighthill[1972].

be precise-- has done it wrong. On the top of pg. 115 is clearly stated, that after the four lines of code given above a case analysis is done. "The case r = w, where colr may have one of three different values, reduces to the alternative clause of the earlier program but for the fact that the "buck:swap(r,w)" -- r and w being equal to each other-- can be omitted." You have just omitted that. Lower I have written "Then the three alternatives can merge and a single text deals uniformly with the second pebble, the colour of which is still given by colr." You allow yourself to break open the then following eight lines of code, allowing a fourth control path to merge -- when there was no "second pebble". I conclude that on line 4/5 of page 5 of your manuscript "with bugs removed" had better be replaced by "with bugs inserted". I enclose for your benefit an annotated version of how the "final composition of our second version" could look like. The repeated references in your text such as "incorrect", "a failure", "contains bugs", "makes a mistake"-- that I made an error seem therefore all misplaced. I am not used to authors blaming *me* for their own blunders!

Allow me a few more remarks about your text. On page 4, line 6, you have "He claims to have used the following thought process". My limited knowledge of English may mislead me, but in my ear that sentence sounds as if you cast doubt on my truthfulness, and I am unaware of having given any reason to do so.

In the footnote on page 6, "occurred" and "occurring" are misspelled.

On page 9, "*goto* skip whites" is executed with colr != red; therefore "goto make swaps" is executed only with colw = red; doesn't this imply w > r? Does w > r not also follow from the fact that "*goto* make swaps" is from within a for-clause "*for* w := w downto r + 1? I don't understand the need for the "*if* w > r *then*" on the last line but 4 of your program.

I am glad that you have followed my advice on page XV "it gives you the opportunity to compare your own solution with mine; and it may give you the satisfaction of having discovered yourself a solution superior to mine." Whether such a discovery is sufficiently significant to justify its publication, is quite another matter, in particular when it concerns an effort that I did not even bother to complete and that was only included to give my readers at least one example of the kind of combinatorial complexity that programmers should avoid. (It was a pleasure to observe, how well that advice of mine served you.).

Your conclusion contains so many misrepresentations of facts that it is unacceptable. And the last sentence raises the question whether there is any solid meat in your conclusion "that programming is a much more complex and creative activity than people currently believe". To which people are you referring? There are so many simple-minded people who are not worth the trouble of contradicting them.

As a final piece of advice: it is a bad habit to rush into publication and to submit a paper when the ink has hardly dried.

3.3 Second Version Submission Letter

From: Author

To: Editor 1

Date: March 16, 1977

I have enclosed a new version of my paper which corrects some errors and, I hope, expresses my position more clearly. I sent an earlier draft to a number of people and have attempted to incorporate answers to their criticisms. Your point about the unclear nature of refinement is well taken, and I have included a definition from *Notes on Structured Programming*.[1]

I think data abstraction suffers from the same sort of problem as refinement. The initial selection of high level data structures and their interface routines forces a programmer to think in one specific way. This makes it very difficult to see solutions which involve data structure modifications. I worked with Jay Earley on VERS and had the feeling no one really ever started by choosing abstract data structures but simply used VERS as a language for writing down algorithms (as some people use English) and then wrote programs as people normally do using much lower level languages.

1. Dijkstra[1972a], p. ??.

Finally, I think abstract data structures may hinder the growth of new algorithms. I've been looking at the various sorting programs that led to heapsort (they're all published in the algorithm section of the CACM), and it seems to me the ability to see a heap simultaneously as an array and a tree is crucial. If Floyd had been limited to only abstract data structures and access primitives, it would at least have been much more difficult to discover heapsort.

I appreciate all the time and effort you are putting into editing my paper.

3.4 Second Version Response to Dijkstra

From: Author

To: Professor Dijkstra

Date: April 20, 1977

Thank you for your letter of Jan. 24. I found your comments extremely valuable, but of course don't agree with a number of your conclusions. You are correct in claiming that I composed your program incorrectly. I interpreted your discussion in general terms rather than as a specific recipe for how to compose the program. You seem to switch between the two levels. For example, in the first line of the paragraph on the top of page 114, you make a general comment abut the program's composition. I assumed you meant the first paragraph on the top of page 115 to be taken in the same vein. I now see my error. Also, I assumed you would handle the r = w case in the last case statement of the program since there is no need for special handling of r = w and the code is almost identical. I have rewritten my paper to eliminate any reference to errors and am now using the program you sent me.

I have been surprised by the way people interpret my paper. I meant the comments on the errors, which I see now were imagined, to be supplementary to the main argument of the paper which you, and I must admit a number of other readers, missed. The failure is in the refinement which results in no improvement. I have rewritten the paper to make this point clearer.

People also misunderstand my attitude about errors. I think they are extremely important to the whole programming process. Programming errors provide a way of seeing programs in much the same way symptoms give structure and concreteness to human disease.

I included the comment about your thought process "he claims to have used the following thought process" because on page xiii line 11 you write "I envisage doing so by describing the -- real or imagined -- design process...".[2]

Thank you for pointing out my spelling errors. I have tried to eliminate them.

You are right that the *if* statement 4 lines from the bottom of my program is unnecessary, and I have removed it. It came up because I debugged the program and that statement didn't effect its correctness. Also, your constraints make the addition of extra program statements almost irrelevant.

I have attempted to make my conclusion more 'meaty' and easier to understand. I think we disagree politically and have attempted to show how the political viewpoint implicit in your programming theories leads to worse programs and prevents people from developing to their full potential. My somewhat strong tone is an attempt to make this difference as easy to understand as possible.

I have included the new version of my paper and would appreciated hearing any new comments.

3.5 Dijkstra's Response to Second Version

From: Professor Dijkstra

To: Author

Date: April 26, 1977

I can indeed vaguely recall having commented on a manuscript you mailed me a few months ago. I have not filed that manuscript, nor have I kept a copy of my letter to you; I had --kindly?-- forgotten your

2. Dijkstra[1976].

name and even my diary --in spite of all the irrelevancies it contains-- does not mention your correspondence. From this evidence I must conclude that I must have judged the manuscript totally insignificant, a conclusion which does not seem incompatible with its second version, which strikes me as silly in more than one way.

1. Very little objection, I think, can be based upon a version of a program that an author included in his book for the purpose of illustrating the kind of logical complications a programmer should learn to avoid. (If, in addition, the author's complicated version is not more efficient that his simple solution, you are only strengthening his point.)

2. If you want to make the point that stepwise refinement is not an infallible way of arriving at a good solution, you are stating the obvious. To quote George Polya -- from "How to solve it", second edition, 1956-- under "Rules of Discovery":[3]

> The first rule of discovery is to have brains and good luck. The second rule of discovery is to sit tight and wait till you get a bright idea.
>
> It may be good to be reminded somewhat rudely that certain aspirations are hopeless. Infallible rules of discovery are hopeless. Infallible rules of discovery leading to the solution of all possible mathematical problems would be more desirable that the philosopher's stone, vainly sought by alchemists. Such rules would work magic. To find unfailing rules applicable to all sorts of problems is an old philosophical dream; but this dream will never be more than a dream.

With the above quotation from Polya I am familiar much longer than you can be with your quotation from Feyerabend, and I am not aware of ever having expressed --or even suggested that I held-- an opinion contrary to Polya's. If you want to oversimplify my views first, and then elaborate on the inadequacy of the grotesque distortion, you are free to do so; I don't feel addressed by it any more.

In your letter you state you would appreciate my comments. I have only one advice: unless you want to act like Don Quixote, fighting ills of your own imagination, burn this manuscript, for I cannot see how -- to quote your letter-- "the political viewpoint implicit in my programming theories" and the harm that that invention of your is supposed to do can be the subject of a scientific paper. Yours ever,

cc: [Berkeley Professor]

3.6 Version Two Rejection Letter

From: Editor 1

To: Author

Date: July 15, 1977

I am obliged to inform you that I will not be able to accept your paper, "A Failure of Structured Programming," for publication in the *Communications of the ACM*. The reasons are quite special, and I would like to go into them at some length.

Your letter of March 16 contains, in my opinion, a number of sharp observations about the limitations on structured programming. These go further than the generally accepted observation that structuring is no substitute for the creative process, and shed light on the deficiencies of structured programming in situations in which a naive view would expect it to be the methodology of choice.

An article which elucidated those views and illustrated them with interesting and counterintuitive examples would be most welcome. This, I gather, is what you have in mind with the present article, and the use of one of Dijkstra's examples was well chosen to this purpose.

Obviously, however, a necessary condition on your paper, once you choose this approach, is that your representations of *Dijkstra's* position be beyond criticism. Otherwise regardless of the merits of your development *in extenso*, the paper is open to attack, with at least partial justification, as tilting with

windmills.

My personal view is that you are on the right track, but that any future paper - which I encourage you to submit - must at least contain a flawless and undisputed presentation of the existing work you propose to use as a jumping-off point.

cc: [Berkeley Professor]

3.7 Rejection Appeal Letter

From: Author

To: ACM Official 1

Date: August 29, 1977

My paper entitled "A Failure of Structured Programming" was recently rejected by the Programming Techniques Department of the Communications of the ACM. The paper examines a solution to the Dutch national flag problem published in *A Discipline of Programming* by Edsger Dijkstra in which an attempted efficiency improving refinement is show to produce no efficiency gain. I am writing to you as Chairman of the Publications Board because of my concern that the paper may not have received fair review.

Manuscripts involving methodological issues are normally edited by [Berkeley Professor], but since I know her personally, I asked [editor 1], the other co-editor, to edit it. He agreed and has done an excellent job in the sense that he conscientiously acknowledged my communications and edited the paper promptly. My concern involves the grounds and manner of the rejection. There is one additional point which may be relevant. An early version of this paper contained a mistake in which I miscomposed and thereby discovered an imagined error in Professor Dijkstra's refined solution. I believe that since I have removed my error, it is no longer a point of contention.

As I read [Editor 1's] letter, the rejection occurred because:

1. It is necessary that my representation of Professor Dijkstra's position be beyond criticism, and
2. My representation is wrong and in fact a "tilting with windmills" (attacking a position which Professor Dijkstra doesn't hold).

Since no referee's report or other detailed analysis was included with the letter of rejection, I can only speculate on the nature of my purported misrepresentations. In addition, since I provided detailed references to back up my claims, I was surprised that the refusal contained no mention of those references.

My concern is based upon the remarkable similarity between the objections Professor Dijkstra used in his response to my paper and those upon which [Editor 1] states his decision was based (see enclosed letters). Because it is unlikely that two independent responses would both invoke the image of Don Quixote and tilting with windmills, it occurs to me that the rejection may have been based on Professor Dijkstra's letter. It seems unfair to allow the founder of a theory to referee a paper which attempts to argue against his theory. One would expect Professor Dijkstra to find my paper unconvincing, but he shouldn't be able to prevent its publication. If there is some other basis for [Editor 1's] arguments, I would appreciate an opportunity to respond.

My concern is heightened because the claims in Professor Dijkstra's letter are inconsistent with his discussion of the Dutch national flag (DNF) problem in *A Discipline of Programming.* He states that since the DNF problem was included not to illustrate refinement but to illustrate "the kind of logical complexity a programmer should learn to avoid," no methodological objections can be based on it. But, he explicitly uses refinement (p. 114)[4] to produce what he believes is an "elegant solution" (p. 117). He is also specifically concerned with improving his program's efficiency through refinement (p. 114) but was completely unaware that the refined solution was no better than his first solution using his efficiency criteria (p. 116). Finally, his claim that the failure of refinement strengthens his point about avoiding logical

4. Dijkstra[1976].

complexity cannot be correct since he believed his refinement succeeded (p. 117).

I don't believe I distort or oversimplify any of Professor Dijkstra's views. I understand, and never state otherwise, that Professor Dijkstra believes refinement is sometimes fallible. My contention is not that refinement is a basically good method which can go wrong, but it is actually detrimental to the practice of programming and should not be used or taught. Refinement is such a basic part of the fabric of thought that making it into an explicit method leads to less problem specific understanding and therefore worse programs. The problem specific understanding then guides the actual programming. Of course, there are many ways of understanding, and many program styles consistent with a given analysis. From a psychological viewpoint, refinement creates the illusion that there are general methods when none exist. My argument for this point is based on the paradigmatic nature of the DNF problem because Professor Dijkstra, a founder of structured programming (SP), erroneously applies it to a simple problem which he created and which involves a clear application of refinement. I would gladly modify my paper to improve the clarity of this point.

There is a view of science due to Carl Popper which identifies a good scientific theory as one which makes claims (called falsifiers) which are basic to the theory and open to refutation. I think it is important that my paper be published because of structured programming's (SP) lack of any openly announced falsifiers. Successful applications seem to count as positive evidence, but unproductive applications are attributed to the individual programmer's lack of skill. My paper only casts doubt on SP, but due to the almost universal acceptance of SP and due to the dearth of published criticism, it is important that the programming public be given a chance to evaluate any negative evidence. I have included a quotation which discusses the difficulties involved in criticizing Freudian Psychoanalysis because I believe it applies with change to SP.

It is important that my paper appear in the CACM rather than some more technical journal because the paper is intended for a wide audience. The Dutch national flag problem is easy to understand, requires no mathematical background, and results in short programs.

If Professor Dijkstra, or anyone else, believes my paper contains faulty arguments, I would be happy to have his or her rebuttal published along with my article.

I appreciate your consideration.

3.7.1 Professor Cioffi's Quotation

> It is characteristic of a pseudo-science that the hypotheses which compromise it stand in an asymmetrical relation to the expectations they generate, being permitted to guide them and be vindicated by their fulfilment but not to be discredited by their disappointment. One way in which it achieves this is by contriving to have these hypotheses understood in a narrow and determinate sense before the event but a broader and hazier one after it on those occasions on which they are not borne out. Such hypotheses thus lead a double life--a subdued and restrained form in the vicinity of counter-observations and another less inhibited and more exuberant one when remote from them. This feature won't reveal itself to simple inspection. If we want to determine whether the role played by these assertions is a genuinely empirical one it is necessary to discover what their proponents are prepared to call disconfirmatory evidence, not what *we* do.[5]

3.8 Appeal Response

From: ACM Official 1

To: Author

Date: September 6, 1977

Thank you for your letter of August 29 concerning the decision regarding your manuscript "A Failure of Structured Programming."

I am turning this correspondence over to [ACM official 2], Editor-in-Chief of Communications, so

5. Cioffi[1974], p. 474.

that he may look into this matter. Both as wearer of the latter hat, and of that of Chairman of the Editorial Committee, he is the appropriate individual to be directly concerned with the issues you raise, although of course I should be pleased to conduct a further level of review should that ultimately become necessary. I am sure that he will be back in touch with you shortly with his findings.

3.9 Rejection Explanation

From: Editor 1

To: Author

Date: September 6, 1977

In response to your letter to [ACM Official 1] concerning my rejection of your article, "A Failure of Structured Programming," I feel that I owe you two things: (1) an apology for certain failures in the editing process, for which I am solely to blame; and (2) a much fuller account of the reasons which are pertinent to the rejection of the *most recent draft* of your article.

With regard to (1): In looking over my file (by now quite extensive), I discovered no letter in which I included copies of referees' reports. These conceivably may not have been sent; if so, it was a clerical error which I should have caught. In fact there were two such reports, which are included. The other point I should mention is that soon after sending you the letter of rejection, it occurred to me that my invitation to submit a new paper might be construed falsely to mean that it could not be based on the Dutch National Flag problem, or on Dijkstra's handling of it. I regarded this as sufficiently serious that I attempted to call you at Berkeley several times. At length I left a telephone message to that effect. I sincerely hope that it go to you.

My insistence on accuracy in representing Dijkstra's position was based on my reading of the *first* draft. On careful reading of the second draft, I now concur with you that this point was successfully dealt with, and that its accuracy in dealing with Dijkstra's claims was - with one significant exception, to be explained below - quite sufficient. In addition, the misplaced emphasis in the first version on a bug in Dijkstra's program was also corrected, so that the focus was no longer peripheral. I believe that Dijkstra's insistence that was no essential difference between the two drafts was a factor in my mistaken emphasis on accuracy of representation.

Now to the more substantive issue (2):

I have now read Dijkstra's and your article carefully, so that I think I can comment on matters of content without undue influence from referees or from consultants.

I regard Dijkstra's article as heavily flawed. The essential flaw is that, having set himself the goal of constructing a (swapwise) *optimal* algorithm for solving the DNF problem, he then constructs an obviously correct outer skeleton for the *unoptimized* DNF problem, and then proceeds to a series of "observations" that are nothing more than heuristics for possible improvement in the algorithm's average performance. Not surprisingly, these heuristics are not all improvements. Even less surprisingly, they do not in tandem achieve an optimal algorithm. In fact not a shred of argument is ever offered that (1) any of them belong in an optimal refinement, or (3) *any* refinement of his skeleton exists that will achieve an optimal algorithm.

The very real objection exists, therefore, that Dijkstra's "refinements" are not refinements at all but simply bad heuristics, unjustified, which he inserts into has program by "refinement".

If this analysis is right, your attack on Dijkstra's paper is *not* an attack on programming by refinement, but merely an attack on a bad exposition of it.

Nevertheless, there is a seed for a most attractive paper, which I would look very kindly on: A paper in which it is shown that, plausible though Dijkstra's skeleton is, it is a blind alley in that either (a) no refinement will achieve an optimal algorithm, or (b) only a refinement of excruciating complexity will work. I rather suspect that formulation (a) is right.

However, that is not what you do. Rather starting from a new tack, you first "discover" an optimal solution for the two-color flag problem. This solution, from what I can see, is none other than separation by radix exchange, which has been known for at least 20 years (see Knuth, Vol. III; you should mention

that is is an old problem).[6] By an ingenious generalization, you produce a three-color variant which is quite elegant. You then make the valuable point (which should be more explicit) that extension is itself rarely expressible as refinement, because the changes are interlaced in the original. Now comes the rub: By careful argumentation (*viz.* McMaster's article on the same subject),[7] it turns out that algorithm is optimal.

Yet the requirement of optimality never is a clear or demonstrable element in your methodology, granted that it is not refinement. It proves on *subsequent analysis* to yield an optimum solution. Therefore, as far as concerns your methodology of *construction,* it is as heuristic as Dijkstra's. The reader is thus left with both the feeling that the driving idea(s) behind your algorithm and Dijkstra's are *both* rather opaque.

It should be clear that a new paper on the same subject might work several new angles: For instance, (1) that problems posed in which there is a requirement for optimality or near-optimality have much less chance at realization through refinement than problems without such constraints, (2) that problems with optimality or near optimality requirements may be achievable only with colossal difficulty via refinement (since this forces the verification process to be applied to the part of the program already constructed), (3) that plausible skeletons of the programs in an early state of refinement which easily yield nonoptimal solutions may be *entirely wrong* when optimality requirements are added, and (4) that the process of extension is not in general expressible as refinement.

These, as I see them, are some valid ideas, although they do not constitute a universal attack on structured programming. One of the weakest aspects of your paper was its overgeneralized claims about the invalidity of structured programming, which was in my judgement not borne out by the paper. This was the primary reason for its rejection, which stands.

If you follow the spirit of these suggestions, I believe you will produce a paper with a good chance of publication.

3.10 Response to Rejection Explanation

From: Author

To: Editor 1

Date: October 19, 1977

I appreciate your prompt response to my letter to [ACM Official 1]. I now understand your position much more clearly but still think "A Failure of Structured Programming" was unfairly rejected. I did receive your telephone message and appreciate your concern but believe your suggestions for a new paper involve a misunderstanding of refinement and the nature of SP. Your new review seems also to be based on that misunderstanding.

Optimality is not an issue in either my argument against refinement or in Professor Dijkstra's treatment of the DNF. My three color solution is not optimal because increasing the number of pebbles inspected before doing any placing reduces the average number of swaps at the cost of increased program size and complexity. I do not know whether the improvement is a fixed constant or is actually proportional to the number of buckets. In the limit, this strategy has the effect of removing the requirement that each pebble be inspected once. Both Professor Dijkstra and I believe program choice must be based on many criteria, not just optimality, and "as conscientious programmers we should investigate how complicated a possibly more refined solution becomes." (Discipline, p. 114). Finally, my argument in no way hinges on the fact that the failure occurred in an efficiency improving refinement. Professor Dijkstra claims he is using "refinement" not "efficiency improving refinement" or "optimality producing refinement".

Refinement is a heuristic method, and therefore it can never be disproven (see Dijkstra's quotation of Polya in his letter to me or his 1975 Pacific ACM address).[8] Since refinement (and SP) are heuristic, they may always be used successfully by a sufficiently skilled programmer. It follows from this that my solution to the DNF problem could be discovered in many ways (exhaustive case analysis, good intuition, or whatever). I included the way I happened to discover it to show an example of an approach outside of SP.

---

6. Knuth[1973], pp. 123-129.
7. McMaster[1979], pp. 842-846.
8. Dijkstra[1975], pp. ??

The point I try to make is that problem solving is by its very nature opaque and therefore universally applicable heuristics (let along algorithms) are illusionary. I believe my analysis of the DNF problem provides evidence for exactly this point. An educational approach to go along with my view would expose students to as many programming styles and methods (tricks) as possible and encourage them to adopt their own approach.

I find it rather strange that you suggest Professor Dijkstra may not have been using his own methodological rule correctly in spite of his claim that he is illustrating its use (p. 114).

I am still concerned about two matters:

1. Both enclosed referee's reports apply only to my first version and do not seem to have been involved in your editorial decision. The double spaced report only discusses the error which has been corrected in the second version. The single spaced report, which was written by one of friends from Berkeley, Colin McMaster, was not typed unit July 15, and was mailed some days later Yet, your rejection letter was also dated July 15.
2. You seem to base part of your decision on your lack of agreement with my conclusion. This is a decision which is usually left up to the reader. If the policy of not publishing material which fails to convince an editor had been followed throughout the history of science, many of the theories we now believe would never have been published.

P. S. By the way, the two color problem actually appears as an exercise in Knuth Vol. 3 (5.2.2-33).[9]

3.11 Third Version Submission Letter

From: Author

To: Editor 2

Date: February 8, 1978

I submitted a paper entitled "A Failure of Structured Programming" to the Programming Techniques Department of the CACM. This paper was edited by [Editor 1]. I spoke with him on Friday, January 13th, and he made a number of specific suggestions which he felt would make my paper suitable for publication and also suggested you take over as editor. He said he would get in touch with you and describe the situation. I agreed to do the following things:

— Send you copies of our previous correspondence.

— Include copies of both the old and new versions of my paper

— Modify my paper to eliminate certain problems and describe in this letter those modifications.

The remainder of this letter discusses his three specific objections and describes my changes.

1. [Editor 1] felt my paper's tone sounded too much like a personal attack on Professor Dijkstra.

   I made the following changes:

   a. Use of Professor Dijkstra's title in a few places.
   b. Changed a number of sentences discussing the failure from "Dijkstra <something>" to "The failure involved <something>."
   c. Gave Professor Dijkstra explicit credit for the observation in footnote 3 in [Chapter 2 section 4].
   d. Removed a number of critical adverbs (e.g. completely wrong to wrong) and strengthened positive adverbs (e.g., alright to good).

This paper still is not written in the normal detached scientific style because one of my points is that

9. Knuth[1973], p. 137.

programming is a personal activity which involves matters of individual taste.

2. He felt my claim that all types of refinement were similar was wrong.

   I think there is considerable evidence that both Professors Dijkstra and Wirth view all types of refinements as one process and I have included as many of their discussions on the nature of refinement as I could find (see enclosures).[10] Refinement seems never to have been explicitly defined, but no distinction is made in discussing the various types. Also, Professor Dijkstra never objected to my claim that the types of refinement are similar in his correspondence concerning my paper.

3. Finally, he felt my conclusions were stronger than my evidence warranted.

   I've rewritten my conclusion to state my argument more clearly and also explicitly state my conclusion is only my opinion. I feel the reader ought to have a chance to judge for himself my conclusion on structured programming.

3.12 Third Version Rejection Letter

From: Author

To: Editor 2

Date: October 5, 1978

I have received the enclosed referee's reports on your paper, "A Failure of Structured Programming," which you have submitted for possible publication in CACM. On the basis of these reports, I have decided that your paper is unfortunately unacceptable for publication in CACM. In view of your feeling your paper was treated unfairly under the previous editor, I would like to elaborate on this decision.

First of all, the two referees who examined your revised manuscript are people who have thought deeply about programming methodology, and who are well respected in this field. I trust their judgements in this area very much more than I trust my own. However, I would be willing to send out your manuscript for even a third opinion, if you feel there is any reason to do so. (Of course, such a third opinion would have to be very strongly in your favor in order to induce me to change my decision.) In any case, I offer you this option if you feel that the referee I chose were in any way unfair.

My editorial decision was based primarily on the reports of the referees. I would like to "change hats" from "editor" to that of "interested computer scientist" in what follows, in order to give you some feedback and comments of my own.

You might be interested to know that a practical problem essentially identical to that of the DNF problem arose once in my own research. Bob Floyd and I had developed a very efficient algorithm for finding medians (see CACM 18 (March 1975), pp. 165-172 and the algorithm SELECT in the same issue.) The innermost loop of this program involved partitioning a set X of numbers into three subsets A, B, and C by comparisons with elements u and v where u < v:

```
A = {x in X | x <= u},
B = {x in X | u < v and x <= v},
C = {x in X | v < x}
```

Where X is kept in any array, the sets A, B, and C correspond exactly to the red, white, and blue pebbles of the DNF problem.

All of my initial solutions followed the DNF rule of never examining an element more than once. I was disappointed to find our algorithm in these cases always seemed slower than a previously published algorithm due to Hoare (CACM 4 (July 1961), p. 321). Hoare's inner loop involved choosing a random element u from X and then dividing X into the two sets:

A = {x in X | x <= u}, and
B = {x in X | u < x}.

10. The material enclosed was ??.

The algorithm Hoare presents for his inner loop is the same as you suggest for the two-color case. (Unfortunately, I no longer have copies of my initial programs available).

Eventually, however, we were able to transform the theoretical superiority of our procedure over Hoare's into a practical superiority as well, by finding the most efficient program for our inner loop. Although it now violates the constraints Dijkstra imposed for solutions of the DNF problem (that is, we will look at pebbles more than once), it seems to provide the most efficient solution when the actual running time is to be minimized.

Our solution is:

(Step 1) Use the efficient two-color algorithm to separate the red pebbles from the "white and blue" pebbles. (That is, treat white and blue as equivalent colors in this step.)

(Step 2) Use the efficient two-color algorithm to separate the white pebbles from the blue ones.

This solution seems to outperform any solution which adheres to the DNF problem constraints. This fact is perhaps unintuitive at first glance, and was only discovered after quite a bit of experimentation. The average number of swaps turns out to be $2N/9 + N/6 = 7N/18$, considerably less than your solution.

I'm not sure what relevance all of the above has to your paper, except to point out that something like the DNF problem arises in practice.

Now to other remarks about your paper.

I agree with your basic point that structured programming (or refinement) is unlikely to lead to optimal programs. When efficiency of execution is the ultimate objective, one must be prepared to sacrifice readability of code and the luxury of writing the program in a single sequence of refinements. A large number of completely worked out possibilities must be analyzed and compared; subtle details can have great effects on the overall speed. And if one is really interested in *optimal* (rather than just good) programs, an equal amount of effort must be spent proving a lower bound on the achievable running times (as McMaster has done for the DNF problem). Developing optimal programs often depends on tricks and "*ad-hocery*".

The major problem in software methodology today seems to be the development of techniques which enable the programmer to handle very large, complex problems and to produce readable, maintainable code. Efficiency of execution is much less important and is becoming less so as hardware speeds increase. Structured programming seems (to me) to be a valuable technique for attacking a very large, complicated problem. If it produces a very efficient solution as well, that's all the better, but the important point is that it enables one to find a working, correct, readable solution. I think Dijkstra's chapter on the DNF is misleading since the problem is so simple that one can afford to pay a lot of attention to efficiency without paying undue costs in terms of complexity. The fact that one can prove that some program is, in fact, optimal for the problem only demonstrates to me that the problem is really trivial and a toy; structured programming perhaps is most useful on problems that are so large that they have no hope of having a provably optimal solution.

My own basic reaction to your paper is that while I find it amusing to see that you've more or less caught Dijkstra doing useless or even wasteful refinements to his program, your paper tries to get too much mileage out of such a simple example. The technique of refinement is a tool for program design. Although any tool can be used for the wrong purposes (and using refinement to find optimal programs may be like using a saw to dig for gold), and although even a master can perhaps misuse a tool, these facts do not really cast much double on the fundamental utility of the tool.

Your presentation would be much more effective if you were to err in the opposite direction, and understate (rather than overstate) the implications of your observation for the utility of structured programming. A "hard sell" here is easily taken as a basis for dismissing your paper.

In any case, the above remarks and suggestions are merely my own personal comments The editorial decision not to publish your paper was based on the referee's reports.

Please feel free to contact me if you have any questions or wish to discuss this decision. In view of

your (somewhat justified) complaints concerning the previous treatment of your paper, I hope to satisfy you that your recent revision was given a fair and competent hearing.

3.12.1 Referee's Report 1

Article: A Failure of Structured Programming

Author: Steven Meyer

Thank you for sending me "A failure of structured programming". I would recommend you not publish it.

I have seen many attempts to pull Dijkstra off the high pedestal on which the world has placed him, but this is one of the silliest. To reject a whole methodology on the basis of this one trivial example is absurd. Obviously the technical discovery reported in the paper is worth a brief publication, and I look forward to seeing McMaster's paper, which appears to treat the matter thoroughly and scientifically.

I am very much in favour of publication of controversial papers, especially ones which attack an "accepted orthodoxy". But they should be based on a little more than a rather minor improvement to a rather minor example program.

The quotation from Feyerabend is also silly. Who ever would propose that "science" should "run according to fixed and universal rules"? Science shouldn't be "*run*" at all. Scientific *research* should be *conducted* in conformity with general *standards* of scientific reasoning. Scientific *discovery*, of course, is unfettered, because it is essentially based on "informed guesswork". But that does not absolve the scientist of responsibility to substantiate his "guesses" in a most rigorous fashion.

3.12.2 Referee's Report 2

Article: A Failure of Structured Programming

Author: Steven Meyer

This paper has relatively little technical content. It does have a point to make, namely the use of stepwise refinement does not guarantee good programs. However, I'm not convinced anyone really believed the opposite. I believe the paper is of border line quality for publication, especially in the CACM. I'm undecided about publishing but leaning toward negative.

If it is published, it should be slightly revised. First, it is not true, as the author states on page 2, that stepwise refinement led to the poor second solution. Stepwise refinement just didn't guarantee that the better solution would be found. This misstatement must be corrected.

Second, it might be useful for the author to restructure has program, since his use of goto's raises extraneous issues anovated with the trivial meaning of "structured programming" (i.e. no goto's). I am enclosing an alternative program.

Chapter 4

Should Computer Programs be Verified

4.1 SIGSOFT Letter

Professors De Millo, Lipton, and Perlis recently published a paper which questions the value of program verification.[1] The appearance of this paper is interesting because the idea that programs should be proven correct rather than debugged has had near universal acceptance.

A critical commentary on this paper written by Professor Dijkstra was published in these notes (April 1978).[2] I don't wish to discuss the details of the disagreement here but rather to suggest that professor Dijkstra's commentary contains two empirical (quasi-empirical) claims which may provide the basis for a scientific test of the utility of program verification.

There is a view of science first proposed by Popper that any true science must contain open and testable claims[3] Lakatos has shown that this same testability applies to mathematics in the form of methods (approaches) which are tested by evaluating their problem solving success.[4] My intention in pointing out the following two claims is to encourage work which will lead to such a scientific test of program verification.

In response to the De Millo et. al. argument that program correctness proofs are inherently long and complex and therefore unconvincing, Professor Dijkstra agrees that long format proofs are unconvincing but states

> It is the mathematician's task to arrange his arguments in such a fashion that avoidable formal manipulations are, indeed, avoided, and to discover those theorems that do admit to a concise proof.[5]

If it turns out that no one is able to find concise (and elegant?) proofs of programs written in the everyday practice of programming, it would cast serious doubt on program verification. On the other hand, the publication of such concise verifications would lend strong support to program verification. It occurs to me that *Software Tools*[6] might contain programs which are difficult to prove concisely. Of course, it may turn out that concise verifications are possible for certain applications or situations but not for others.

The question of the existence of concise verification may also be open to theoretical analysis. One approach involves analyzing the inherent complexity of computer programs.[7] Another approach involves the inherent complexity of various formal deduction schemes.[8] Both approaches use artificially constructed problems designed to simplify the mathematical analysis and therefore may not be relevant to real computer programs. Also, when people verify programs, they may not be limited by formal approaches and can rather use their intuition and good fortune to discover concise proofs.[9]

The second claim pertains to the value of formal versus informal reasoning. Professor Dijkstra writes

> Eventually a nice formal treatment is always the most concise way of capturing our understanding and the most effective way of conveying the argument with all its convincing power to someone else.[10]

The publication of formal treatments for things which seem difficult to treat formally would provide impressive positive evidence for program verification. I would be especially impressed by a formal treatment of the U.S. tax code which is more effective (and concise?) than the current combination of

---

1. De Millo[1977], pp. 206-214.
2. Dijkstra[1978], pp. 14-18.
3. Cf. Popper[1959] for the original argument and Cf. Lakatos[1970], pp. 91-196 for a further development of the viewpoint.
4. Lakatos[1978], pp. 24-42.
5. Dijkstra[1978], p. 14.
6. Kernighan[1976]
7. Cf. for example Jones[1977], pp. 338-350.
8. Cf. for example Cook[1976], pp. 28-32.
9. Lakatos[1976], pp. 1-5.
10. Dijkstra[1978], p. 14.

statutes and judicial decisions.

The reader should understand that I have chosen to only discuss Professor Dijkstra's commentary because he is the most eloquent and sophisticated supporter of program verification and because I believe limiting my remarks to one commentary simplifies the reader's task.

4.2 Postscript

Section 4.1 was written as a reply to the first version of the De Millo et. al. argument (see above) that program correctness proofs were not successful for behavioral reasons. I meant to argue that verification was also impossible for scientific reasons and that no social change in the practice of programming would make correctness proofs more valuable. This argument was partially incorporated in the much improved second version of their paper,[11] but their main argument remained sociological. An extensive and in the main highly favorable group of responses appeared in the CACM forum.[12] The favorable responses also disagreed with the sociological rather than scientific nature of their argument. In any event and for whatever reasons, most computer science researchers are now convinced that program verification is unpromising.

There is one more argument against verification which was not included in section 4.1 because of space limitations. The argument goes: program proofs are not useful since the program itself already plays the role in computer programming which the proof plays in mathematics. In mathematics, a theorem is stated as a provisional solution to some intuitive problem. The solution (theorem) is then improved by the proof process.[13] One might say the proof and the problems which arise in its construction (bugs) serve to move implicit background knowledge into the problem solution.

In programming, the program itself plays the role of both the theorem and the proof. Difficulties which arise in solving some intuitive problem are detected by testing and debugging which then results in improvements to the program and possibly in shifting the original intuitive problem.

This may be the reason programming works well in solving problems which are complicated and detailed but not deep and generalizable. It may also explain why mathematical algorithm analysis is most successful for programs which are abstract and general but not detailed while performance measurements are most successful for programs which are detailed but not deep.

---

11. De Millo[1979], pp. 271-280.
12. Ashenhurst[1979], pp. 621-630.
13. Cf. Lakatos[1976] for a more complete discussion of this view of mathematics.

Chapter 5

A Constructive Placement Algorithm for Logic Arrays

5.1 Introduction

Advances in integrated circuit technology and logic design aids have made 10,000 gate logic arrays practical in 1983, but computer aided physical design must still be improved to efficiently utilize advances in silicon technology. Since most placement and routing algorithms are NP complete (probably not solvable in polynomial bounded time) in their worst cases, progress has been limited by computing time requirements alone.

This paper describes a constructive phase only placement algorithm that is part of LSI Logic's LDS I logic array design system. It is routinely used to place CMOS circuits with up to 6000 potentially usable gates. The algorithm is limited in the sense that it is optimized for one particular array geometry, but it requires no iterative improvement phase and uses only a few minutes of computer time.

The main scientific contribution of this paper is in showing that a constructive placement algorithm with either pre-placement circuit partitioning or iterative improvement is much more promising than currently believed and in offering reasons why this may be so.

5.2 The Basic Placement Problem

The logic array (sometimes know as gate array) placement problem involves assigning circuit elements to physical integrated circuit transistor gate locations. A circuit element is a cell that implements some logic function and ranges from a simple two pin inverter to high level, multi-pin macro functions such as flip-flops or multiplexers.

Following Fiduccia and Mattheyses,[1] problem input is presented as a set of C cells (circuit elements) connected by a set of N nets (signals or wire lists). The point at which a net connects to a cell (internal macro cell) is called a pin and by assumption each cell contains at least one such pin. Also each net contains at least two pins, and each cell is in at least one net.

This formulation implies that the size of a problem is bounded by the number of cell pins P in a given circuit and means any simple operation, such as circuit input, is O(P) (roughly bounded by P) except that some input schemes may require input pin sorting which is O(P LOG P).

Problem input also includes two data structures that are assumed to have been constructed during circuit input. One is a list for each cell giving each net that cell is in. The other is a list for each net giving each cell that net is connected to.

5.3 The CMOS Logic Array Placement Problem

The particular constructive placement algorithm described here is embedded in LSI Logic's LDS II automated design system[2] and is similar to the LTX system[3] in intent. The current algorithm only applies to the 5000 series of CMOS logic arrays.[4] Arrays in the series use two layer metal for separate horizontal and vertical wiring and only one uniform cell placement area. The cell placement area is a rectangle of R cell rows each of which is G gates wide. The total internal array area is R*G. All external (off chip) signals must be routed through I/O pads (package pins).

The 5000 series consists of arrays ranging in size from 800 to 6000 gates.[5] The equal length gate rows are separated by fixed width horizontal wiring channels. For example, the 3200 gate array family member contains nineteen cell rows with each row containing one hundred sixty eight gates. The area between the chip I/O pad region and the internal cell region also contains usable wiring tracks both above and below the cell rows and to the right and left of the row and channel ends. These areas are somewhat larger than the wiring channels.

---

1. Fiduccia[1982], pp. 175-181, and also Schweikert[1972], pp. 57-62.
2. Koford[1981] and LSI Logic[1983a].
3. Perky[1977], pp. 217-255.
4. Shiraishi[1980], pp. 458-464, Kanada[1981], pp. 427-441, and Werner[1982], 206-214.
5. LSI Logic[1983a]

Each horizontal wiring channel currently contains sixteen tracks and there are no over cell row vertical wiring channels. This causes what is known as the inter-channel feed-through problem.[6] It means that feed-through wires must be routed either over unoccupied gate locations or over free tracks inside cells (usually this means not occupied by a pin). The lack of vertical wiring channels implies utilizations higher than seventy to eighty percent are rarely attainable and for circuits with unusually complex wiring (many pins and large nets), utilization can be as low as fifty percent, but wire complexity and feasible utilization estimates are available to the circuit designer.

5.4 Algorithm Description

The algorithm described here is an improved version of the "epitaxial growth" constructive placement algorithm described in Soukup's 1981 overview paper.[7]

As described in Hanan[1972],[8] a constructive placement algorithm selects and places cells according to some evaluation criteria that is defined only in terms of already placed cells and circuit connectivity information. The approach requires a method for selecting and placing starting seed cells, an evaluation criteria for selecting the next cell to place, and an evaluation criteria for determining where to place a cell.

The basic idea is that at any given stage of the placement process, the cell being placed is the one about which the most is known. This means the cell in the net which has the highest percentage of already placed cells. Also when placing a cell, the largest possible sample of connectivity information is used.

This algorithm is termed "epitaxial" since cells "grow" from the edges inward in a planar pattern even though isolated islands often appear early and not necessarily near the chip edges. It is an improvement over the algorithm sketched by Soukup in using a two step seed placement phase and in usually completing the placement of all cells in a net before starting on the cells in the next net.

5.4.1 Seed Cell Selection

The algorithm assumes I/O cell pad assignment has been specified as part of the circuit design. It then uses the pads to place some cells in the internal cell region which are then used as seeds for the rest of the placement. For every net containing a pad cell, the largest non-pad cell in that net is placed as close to the pad as possible using exact wire grid distance. The pad cell containing nets are selected in arbitrary order. Nearly all cells are placed in their true minimum wire distance locations and usually from ten to fifteen percent of the circuit's gates are placed in this phase. The algorithm is normally insensitive to chip I/O pad assignment as long as pads are uniformly spaced around the chip. After completion of seed placement, pad cells are simply ignored.

5.4.2 Next Cell Selection

Cell selection is accomplished by selecting a net and then selecting the largest unplaced cell in that net. The selected net is the one that has the largest percentage of cells already placed. Ties are broken by choosing the largest net. "Largest" means containing the most cells, and if there are still ties, "largest" means occupying the most most total gates. After a net to place is selected the largest unplaced cell from that net is placed and selection process is repeated. In general placing one more cell in the already completed net insures that the same net will be selected again, but sometimes small nets are finished before the current net is completed. An earlier version of the algorithm placed all unplaced cells whenever a net was selected and in all but a few cases seemed to do as well. The idea behind this selection criteria is that the important idea is not how good the next cell placement is in terms of already placed cells but in making sure the next cell has the most possible location information know about it.

5.4.3 Cell Placement

Once a cell is selected the algorithm tries all unoccupied internal array locations (contiguous gate regions) in which the cell will fit. The cell is placed in the location that minimizes the sum of the weighted minimum spanning tree distance for each net the given cell is connected to. In effect, net distance is computed by finding the closest already placed cell in each connecting net.

---

6. Perky[1977], pp. 217-255, and Kanada[1981], pp. 427-441,
7. Soukup[1981], pp. 1281-1304.
8. Hanan[1972], 213-282.

The evaluation function is measured only to gate, not wire grid, distance but otherwise simply minimizes distance weighted by user supplied weights. The weights are based on real wire grid distance but are usually modified to improve wirability. They were discovered through trial and error and seem to increase the frequency of cross channel connections and to reduce the number of cell row feed through wires.

5.5 Pragmatic Considerations

The algorithm has several user selectable parameters that allow compensation for differences between array sizes and individual circuits. They also aid in producing placements that take advantage of the available automatic router's strong points. Circuits are sometimes placed more than once to improve routability.

5.6 Running Time

This algorithm's running time is bounded by the cell placement phase since the cell selection phase simply involves a minimum calculation over the as yet uncompleted nets performed once for each cell. The number of steps required place all cells is loosely bounded by the sum over all cells i of $A*p(c_i)*n_{max}$. Where A is the number of gates in the array, $p(c_i)$ is the number of pins cell i has, and n_{max} is the size of the largest net in the circuit. Since A and n_{max} are constants for a given circuit, the bound becomes $A*P*n_{max}$ where P is the total number of internal macro cell pins in the circuit. Remember that the sum for all cells C of $p(c_i)$ is P. In practice this value is bounded by $O(P^2)$ since the number of gates is usually comparable to the number of pins and nearly all nets contain less than five or ten pins.

The running time for various placements on a 5 mip Amdahl 470 V6 using IBM Pascal with moderate use of inline procedures ranges from 30-45 seconds for 1400 gate arrays to 8-10 minutes for 6000 gate arrays. I/O and data structure construction times are included.

5.7 Results

The algorithm works well in the sense that it produces wire completion results better than those produced by a random initial placement, net span minimization iterative improvement algorithm similar to those described in Hanan.[9] It also produces reasonable wire lengths and spreads wire congestion out uniformly. In the one case is was compared to a manually placed circuit, it produced better wire length but required more manual clean-up routing.

5.8 Discussion

The described algorithm seems to work well because it makes good use of the edge area of the chip while doing a reasonable job of wire length minimization. The idea of epitaxial growth from edges inward may work because it corresponds to the way circuit designers think of their circuits. The problem of array center congestion, as described in Burstein[10] seems to be avoided by using pad net seed cells. Finally epitaxial growth with distance minimization seems to generate more equally good possible cell locations than iterative distance minimization schemes and therefore seems to work well over a large range of different circuit organizations.

The success of this algorithm seems surprising in light of the commonly accepted belief that the most promising placement approach involves the three step process: circuit partition, constructive initial placement, and finally iterative improvement.[11] On the other hand, it is also accepted that placement is not a solved problem and work is continuing both on other placement techniques[12] and on enhancements to iterative improvement.[13] In general the three step approach may be valuable, but in the case of CMOS logic arrays, there seem to be so many conflicting requirements that must be dynamically balanced that no simple mathematical evaluation function may exist, and possibly the best evaluation function may be pad cell driven epitaxial growth itself.

---

9. *Ibid.*
10. Burstein[1982], pp. 265-269.
11. Hanan[1972], pp. 213-282, Hanan[1978], pp. 28-61, and Soukup[1981], pp. 1281-1304.
12. Perky[1977], pp. 217-255.
13. Kirkpatrick[1982], and Burstein[1982], pp. 265-269.

5.9 Postscript - Comparison with ECL placement

The "epitaxial" growth algorithm was modified to place ECL gate arrays[14] but was considerably less successful. This was primarily due to electrical differences and secondarily due to geometric differences between the two array types. The biggest ECL array in production in 1983 had a maximum of only 144 macro cells called half cells. The only other cell size is a full cell (excluding interface and external I/O cells) that consist of two adjacent half cells. This results in almost no wiring channel congestion since most channels contains at least 14 tracks. As compared to CMOS type arrays, the number of internal macro cell pins is less than 15 percent while the number of available wiring tracks is almost 90 percent. The number of cells in an ECL array is small enough to allow more exhaustive search or, for that matter, only semi-automatic placement. There are so many available wiring tracks that there is no reason to spread out congestion.

ECL array macro cells are often used in a different manner than CMOS macro cells. ECL cells are not always limited to one function but instead may consist of some combination of primitive subcells. One common half cell provides two NAND gates, one with three inputs the other with four.[15] This lack of unique cell functionality reduces the value of signal net based placement.

Also wire length along critical paths is much more important in the faster ECL technology. The resistive metal layer delays are only a small percentage of total delays in both cases, but in ECL, circuit speed is a more significant factor.

Finally, for ECL technology macro cell power consumption balancing over various parts of the chip is critical.[16] This almost dictates the use of some type of quaternary partition placement approach that is incompatible with epitaxial growth. (see Kernighan[17] for a binary partition algorithm.)

14. LSI Logic[1983b].
15. *Ibid.*, p. 4.
16. Motorola[1980], p. 2.
17. Kernighan[1970], pp. 291-308.

Chapter 6

Discussion

6.1 Evaluation

In some aspects this thesis research has been successful and others not. Structured programming is definitely not as universally accepted as it was in the middle 1970s. But that change may have had nothing to do with the arguments contained here. There is now strong evidence that structured programming is at least too narrow and rule like, but no programming can be replaced simply by arguments and evidence since a major component of any such methodology is by necessity psychological.

6.2 Subsidiary Results

Two problems which are in themselves unexpectedly interesting have been identified. The Dutch national flag problem lies at the boundary of trivial and difficult problems and turned out to be difficult to analyze mathematically. The gate array placement problem is simple to state and at least in principle solvable by exhaustive search but in practice seems much more difficult. It seems to require both heuristic techniques and problem shifting.

Another unexpected outcome of this research is the illustration of methods useful in the examination of methodology based theories. Classical philosophical arguments have not been commonly used in the arena of modern science but may become an integral part of sciences with competing methodologies. Careful examination of the problems and solutions from research monographs, publication of real dialogues, and the application of problem shifting and splitting all seem seem to have wider utility. This might effectively change science back to natural philosophy, which was the scientific form before the modern scientific revolution, while hopefully retaining the improved modern standards of empirical evidence.

6.3 Future Work

If the proposed pragmatic approach is to prove valuable, it should result in growth in programming knowledge in general and especially in the area of algorithm design. One such recent successful result is improved prime number testing algorithms which are based on more detailed problem analysis. It seems to me one of the most promising practical areas is better compiler code generation programs, especially for specialized types of computers such as vector processors and microprocessors in which even a small improvement in code quality is worth a large amount of effort. Another general area is better problem specific algorithms to automate the physical design of integrated circuits, placement, auto-routing, and mask checking.

Appendix A

Letter to the editor of Infoworld entitled "Treacherous" by Carolyn Chase:

In the Vol. 5, No. 9 issue, your frontpage article entitled "Western Electric may market four micros" states that "it is assumed that the first software for the new systems will be ported in Western Electric's popular Pascal-like language called C."

To state that C is a Pascal-like language is a deception at best. C flies in the face of all software methodology pertaining to language design researched in the last ten years.

C is a hacker's delight and a management and maintenance nightmare.

A portable assembly language, C is not block structured, has no type checking and is cryptic to read (to name a few important faults).

It is interesting to note (but not really surprising) that managers are choosing to program in C, and "old" and unstandardized language, in an industry growing so fast that staff turnover is very high and the schools cannot possibly meet the demand for the skilled talent. Of course, this also underlines the requirement for portable (i.e. reusable) code which is a major force for many turning to C over other choices such as Pascal, Modula-2 or even Ada.[1]

1. Chase[1983], p. 28.

References

Ashenhurst[1979] Ashenhurst, R. L. (ed.) Comments on social processes and proofs. *CACM*. 22, 11(1979), 621-630.

Bitner[1982] Bitner, J. R. An asymtotically optimal algorithm for the Dutch national flag problem. *SIAM J. Comp.* 11, 2(1982), 243-262.

Burstein[1982] Burstein, M., Hong S. J., Nair, R. Spatial distribution of wires in master-slice VLSI. Proceedings IEEE International Conference on Circuits and Computers, 1982, 265-269.

Cioffi[1974] Cioffi, F. Freud and the idea of a pseudo-science. in Borger, R, and Cioffi, F. (eds.) *Explanation in the Behavioral Sciences*, Cambridge University Press, 1974, p. 474.

Cook[1976] Cook, S. A. A short proof of the pigeon hole principle using extended resolution. SIGACT News (ACM) 8, 4(1976), 28-32.

Cote[1980] Cote, L., and Patel, A. The interchange algorithms for circuit placement problems. Proceedings 17th Design Automation Conference, 1980, 528-534.

Demillo[1977] De Millo, R. A., Lipton, R. J., and Perlis, A. J. Social processes and proofs of theorems and programs. Conference Record ACM Symposium on Principles of Programming languages, 1977, 206-214.

Demillo[1979] De Millo, R. A., Lipton, R. J., and Perlis, A. J. Social processes and proofs of theorems and programs. *CACM*. 22, 5(1979), 271-280.

Dijkstra[1972a] Dijkstra, E. W. Notes on structured programming. in *Structured Programming.* Dahl, O. J., et al., Academic Press, 1972.

Dijkstra[1972b] Dijkstra, E. W. The humble programmer. *CACM*. 15, 10(1972), 859-865.

Dijkstra[1975] Dijkstra, E. W. Craftsman or scientist. Proceedings of the 1975 Pacific ACM Conference, San Francisco, 1975, 217-223.

Dijkstra[1976] Dijkstra, E. W. *A Discipline of Programming.* Prentice Hall, 1976.

Dijkstra[1982] Dijkstra, E. W. *Selected Writing on Computing: A Personal Perspective.* Springer Verlag, New York, 1982.

Dijkstra[1978] Dijkstra, E. W. On a political pamphlet from the middle ages. SIGSOFT Software Engineering News (ACM) 3, 2 (1978), 14-18.

Feyerabend[1975] Feyerabend, P. *Against Method.* Humanities Press, London, 1975.

Fiduccia[1982] Fiduccia, C., and Mattheyses, R. A. A linear-time heuristic for improving network partitions. Proceedings 18th Design Automation Conference, 1982, 175-181.

Floyd[1979] Floyd, R. W. The paradigms of programming. *CACM*. 22, 8(1979). 455-460.

Gotshalks[1978] Gotshalks, G. J. Analysis of the Dutch national flag algorithm and refinements. Computer Science Research Report, York University Downsview, Ontario Canada, December, 1978.

Hanan[1972] Hanan, M., and Kurtzberg, J. Placement techniques. In M. Breuer, (ed.) *Design Automation of Digital Systems, Vol. 1*, Prentice Hall, 1972, 213-282.

Hanan[1978] Hanan, M., Wolff, P., and Agule, P. A study of Placement techniques. *Journal of Design Automation & Fault-Tolerant Computing.* 2, 2(1978), 28-61.

Hoare[1962] Hoare, C. A. R. Quicksort. *Computer Journal*. 5, 1(1962), 10-15.

Jonassen[1977] Jonassen, A. Analysis of the "Dutch flag problem". Institute of Informatics Research Report, University of Oslo, Norway, 1977.

Jones[1977] Jones, N. D. and Muchnik, S. S. Even simple programs are hard to analyze. *J. ACM* 2, 24(1977), 338-350.

Kanada[1981] Kanada, H., et. al. Channel-order router--A new routing technique for a masterslice LSI. *Journal of Digital Systems* 4, 4(1981), 427-441.

Kernighan[1970] Kernighan, B. W., and Lin, S. An efficient heuristic procdure for partitioning graphs. *Bell Systems Tech. J.* 49, 2 (February 1970), 291-308.

Kernighan[1976] Kernighan, B. W. and Plauger P. J. *Software Tools.* Addison-Wesley, 1976.

Kirkpatrick[1982] Kirkpatrick, S., Gelatt, C., and Vecchi, M. Optimization by simulated annealing.

IBM Research Report, RC 41093, April 1982.

Knuth[1973] Knuth, D. E. *The Art of Computer Programming, Vol. 3, Searching and Sorting*, Addison Wesley, 1973.

Koford[1981] Koford, J., and Jones, E. A development system for logic arrays. Midcon (Middle Western Electronics Conference), 1981, Chicago.

Lakatos[1970] Lakatos, I. Falsification and methodology of scientific research programmes. In I. Lakatos and A. Musgrave, (eds.) *Criticism and the Growth of Knowledge.* scientific research programmes. Cambridge, 1970, 91-196.

Lakatos[1976] Lakatos, I. *Proofs and Refutations.* Cambridge, 1976.

Lakatos[1978] Lakatos, I. A renaissance of empiricism in the recent history of mathematics. In I. Lakatos, *Mathematics, Science and Epistemology*, Philosophical papers Volume 2, Cambridge, 1978, 24-42.

Lighthill[1972] Lighthill, J. Artificial intelligence - A general survey. Also known as the Lighthill Report. Cambridge University, July, 1972.

LSI Logic[1981] LSI Logic Corporation, Silicon gate HCMOS macrocell array LSI 5000 series data sheet. September 1981.

LSI Logic[1983a] LSI Logic Corporation, CMOS macrocell and macrofunction library. Revision 4, January 1983.

LSI Logic[1983b] LSI Logic Corporation, ECL macro cell array design manual. Revision A, April 1983.

McMaster[1979] McMaster, C. L. An analysis of algorithms for the Dutch national flag problem. *CACM*. 21, 10(1979), 842-846.

Motorola[1980] Motorola Corporation, Supplementary information to MECL 10,000 macro array preliminary design manual, 1980, 2.

Perky[1977] Perky, G., Deutsch, D., and Schweikert, D. LTX--A minicomputer-based system for automated LSI layout. *Journal of Design Automation & Fault-Tolerant Computing*, 1, 3(May 1977) 217-255.

Polya[1956] Polya, G. *How to Solve it*, second edition, Princeton University Press, 1956.

Popper[1959] Popper, K. R. *The Logic of Scientific Discovery.* Hutchinson: London, 1959.

Shiraishi[1980] Shiraishi, H. and Hirose F. Efficient placement and routing for master slice LSI. Proceedings 20th Design Automation Conference, 1980, 458-464.

Soukup[1981] Soukup, J. Circuit Layout. *Proceedings of the IEEE*, 69, 10 (October 1981) 1281-1304.

Schweikert[1972] Schweikert, D., and Kernighan B. A proper model for the partitioning of electrical circuits. Proceedings 9th Design Automation Workshop, June 1972, 57-62.

Werner[1982] Werner, J. Computer-aided design and design automation for ICs in Japan, 3, 3(May/June 1982) 206-214.

Wirth[1971a] Wirth, N. The programming language pascal. *Acta Informatica.*. 1, 1(1971), 35-63.

Wirth[1971b] Wirth, N. Program development by stepwise refinement. *CACM*. 14, 4(1971), 221-227.

CONTENTS

LIST OF FIGURES