Technical Report CS73003-R

GENERALIZED STRUCTURED PROGRAMMING

Johannes J. Martin

Department of Computer Science

College of Arts and Sciences

Virginia Polytechnic Institute and State University

Blacksburg, Virginia 24061

J. J. Martin

Abstract

In an effort to eliminate some inconveniences connected with Dijkstra's method of Structured Programming, a generalized set of basic flow graphs for structuring programs is suggested. These structures generate the set of all flow graphs that can be fully decomposed by Allen and Cocke's method of interval reduction. It will be shown that programs composed of the proposed basic structures have most, if not all, of the positive characteristics claimed for programs written with the classic rules of Structured Programming. Further, by extending Wirth's programming language PASCAL a set of new control constructs has been suggested that support the proposed set of flow structures.

Keywords: Structured Programming, Flow graph Analysis, Theory of Programming, Graph theory

Introduction

Structured Programming as defined by Dijkstra [4] produces programs whose flow can be described by one of the basic flow graphs shown in Fig. 1.

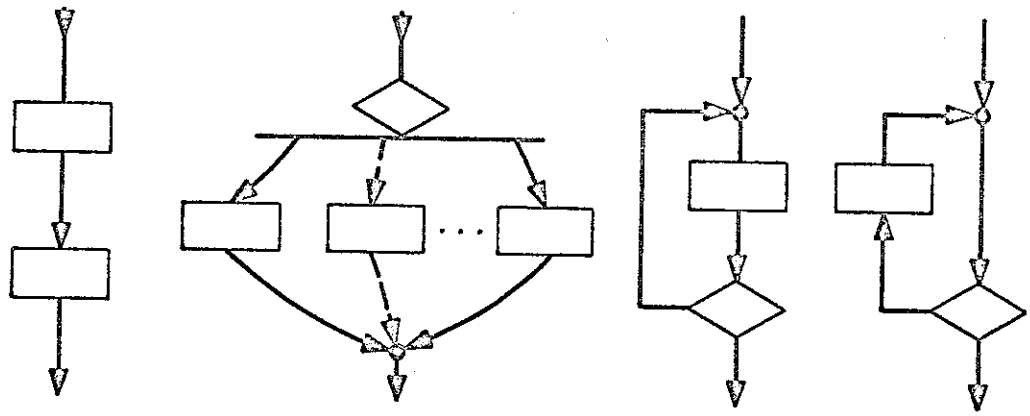


Figure 1 Primitives of Structured Programming

Each square box in such a flow graph represents either a primitive operation or another well structured program. Thus, Structured Programs are recursively decomposable with respect to these basic flow graphs. As a main characteristic each basic flow graph has only one entry point and one exit.

Böhm and Jacopini [2] who have investigated a very similar set of flow graphs have demonstrated that their set, which is even slightly simpler than Dijkstra's, is sufficient for defining all algorithms. Many Programming Languages include control constructs (e.g. the *if-then-else* or the *while* construct in ALGOL 60 and PL/1) that support, to some extent, the use of the basic flow structures of Structured Programming. Wirth's languages PASCAL[11], an ALGOL derivative, provides control constructs for all flow graphs needed in Structured Programming (Table 1) and no additional ones.

```
if<expression>then<statement>[else<statement>]
case<expression>of<const>:<statement>
                    [;<const>:<statement>]* end
while<expression>do<statement>
repeat<statement>[;<statement>]*until<expression>
for<var.>:=<expression><sep.><expression>do<statement>
<sep.>::=to/downto
```

Table 1

Because PASCAL matches the demands of Structured Programming so completely, the notational extensions suggested in this paper will be stated as extensions of PASCAL.

The main assets of Structured Programs are their clarity and relative simplicity. In particular, the method of Structured Programming encourages top down analysis [9] of problems or the development of algorithms by stepwise refinement [10]. Programs developed in this manner are automatically modular; hence, Structured Programming provides a systematic way of modularizing. Furthermore, as Dijkstra [3,4] has pointed out, structured programs display a simple relation between the progress of the computation and the progress through the program text. As a result, the amount of information needed for determining the computational progress accomplished at some point in a program does not depend on the length of the program (since one does not need a trace) but only on the depth of loop nesting and the depth of subprogram referencing at the point of interruption. All these points clearly enhance the transparency of a program.

Nevertheless, many programmers feel that the method of Structured Programming is too restrictive. In particular, since loops can have only one exit (Fig. 1) some simple and very common program structures are outlawed. The classic example is the search loop [6]. The search loop either finds the item wanted, in which case some action A should be taken, or it does not find the item, in which case an alternative action B should be executed. Such a search loop can obviously be interpreted as a program segment that computes a condition and, by virtue of its two exits, selects one of two consequent actions. Thus, such a loop could logically replace a decision box in a flow chart. However, the rules of Structure Programming require that decision nodes are primitive; the substitution of a decision node by a program segment is not permitted since the flow of such a program segment would not conform with any of the four basic flow graphs. Thus, Structured Programming, in effect, classifies conditions which are used to control the flow of a program into two categories:

a) Simple conditions that can be specified in condition boxes because they can be computed by a single expression, and

b) Complex conditions that must be computed in a program segment that precedes the test which ultimately makes the selection because they cannot be computed by a single expression.

This distinction causes programming steps that are motivated solely by structural restrictions imposed by rules of style rather than by the inherent logic of the problem to be solved. At this point, the generally beneficial rules of Structured Programming definitely lower the understandibility of a program.

Shortcomings of Structured Programming have been discussed in the

literature, especially in some articles concerned with the elimination of the goto-statement [5, 7, 13]. However, to my knowledge the problem has never been stated as a problem of discrimination among simple and complex conditions, although it seems to be this discrimination that causes the alledged inconveniences of goto-less or Structured Programming.

In this paper we shall suggest a set of generalized flow structures as a basis for structuring programs. The main idea is to have basic flow graphs with multiple exits so that not only action nodes (square boxes) but also decision nodes can be replaced by program segments. As a result the artificial distinction between simple and complex conditions will disappear.

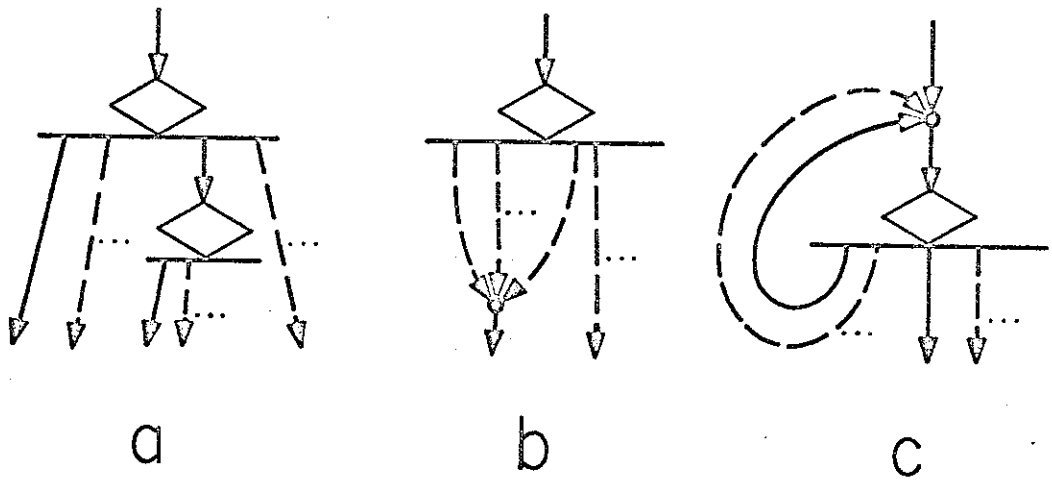
Allen and Cocke [1] have developed a method for analyzing flow graphs which they call interval decomposition. With this method, one can reduce flow graphs that do not contain multiple entry loops to a single node. We shall be able to show that our set of basic flow structures permits the construction of exactly these flow graphs.

We shall further explain why the advantages claimed for the original system of Structured Programming are maintained by the generalized version suggested. Finally, we shall suggest a set of new control constructs (as an extension of PASCAL) which supports our generalized set of basic flow graphs.

Generalized Structured Programming

We define a well-structured flow graph as a graph that can be described by one of the structures shown in Fig. 2 where the nodes represent either primitive operations or well-structured flow graphs. The flow graphs shown in Fig. 2 contain only two types of nodes: nodes with several entry

points and one exit (later referred to as collector nodes) and nodes with one entry point and several exits (later referred to as action nodes).



Primitives of Generalized Structured Programming

Figure 2

Collector nodes do not correspond to any computational action; their function is comparable with that of labels in programming languages. Hence, collector nodes are not further decomposable.

Action nodes have $k \geq 1$ exits and, thus, may occur as decision nodes as well as single-exit action nodes.

Our structures show some resemblance to Dijkstra's sequential (2a), selective (2b), and iterative (2c) modes of operation. However, they are less restrictive since the set of graphs which they generate contains the set generated by Dijkstra's structures as a proper subset. Later we shall refer to our structures as forms 2a -2c; we shall further use the term decomposable (reducible) restrictively for graphs that are decomposable (reducible) with respect to our system.

Properties of Decomposable Flow Graphs

First we shall show that a graph is decomposable if it does not contain multiple entry loops.

1. We shall consider only flow graphs with exactly one entry point. The node R by which a flow graph is entered will be called its root.
2. We postulate that in all flow graphs considered, there is a path R-N for every node N of the graph.
3. Definition: A loop (or strongly connected region) is a set S of nodes such that for each pair of N,M ε S there is a directed path N-M.
4. Lemma: A flow graph that does not contain loops can be reduced to an action node by the forms 2a and 2b.

   (Proof by induction based on 1. and 2. using the fact that the nodes in a loop-free graph (3.) are partially ordered.)
5. Definition: Flow graphs are equivalent, if they can be transformed into each other by collapsing adjacent collector nodes or, reversely, by splitting a collector node into two adjacent ones.[1]
6. Definition: A collector node C is a selection collector if there is a node D such that every directed path C-C contains D and there is a simple path D-C for every arc entering C.

---

[1]Since collector nodes do not represent any action, this transformation trivially does not change any essential property of the flow graph (it is equivalent to the introduction of an additional label for an already labeled point in a program). However, it facilitates the creation of flow graphs for which collector nodes can be classified into

a) selection collectors

b) loop heads for single entry loops

c) loop heads for multiple entry loops.

7. Definition: A collector node that is not a selection collector is called a loop head.

8. Lemma: Every loop contains a loop head. (Follows from 2.,3.,6.,7.)

9. Definition: A loop head L is the head of an S-loop (single entry loop) if there exist paths L-L that are disjoined from all paths R-L except for the node L. Nodes belong to the S-loop if they are contained in such a path L-L. Other loops are called M-loops (multiple entry loops)

10. By splitting the loop head (5.), if necessary, we shall always make sure that all arcs but one entering the head of an S-loop participate in the S-loop. Thus, we shall make sure that only one arc enters an S-loop from the outside.

11. Lemma: If an S-loop L_1 contains the head of an other S-loop L_2, all nodes of L_2 are also nodes of L_1. (Follows from 9)

Note 1: Definition 9. assures that the converse of 11. is not true, i.e. there are nodes in L_1 that are not nodes of L_2.

Note 2: It follows from 11. that an S-loop is not necessarily a proper nest of loops, say, in the FORTRAN sense. Fig. 3 gives an example.

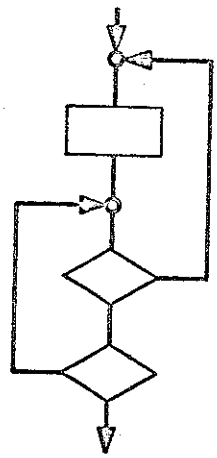


Figure 3 Example of an S-Loop

12. Definition: The rump of an S-loop is the graph that consists of all nodes of the S-loop but the head.

13. Lemma: The rump of an S-loop is a graph with one entry point and one or more exits. (Follows from 12. and 9.)

14. Lemma: An S-loop that does not contain any loop, can be reduced to a action node.

(Proof: The rump is a loop free graph with one entry point (13.) and, hence, reducable because of 4. The loop head can then be attached by form 2c to form a graph with one entry point and one or more exits because of 9. and 10.)

15. Lemma: S-loops that do not contain M-loops can be reduced to an action node.

(Proof: by induction based on 14., partial ordering is assured by 11.)

16. Theorem: Flow graphs that do not contain M-loops are well structured with respect to the forms 2a - 2c.

(Proof: follows from 4. and 15.)

Because the set of flow graphs that can be constructed with the forms 2a - 2c is considerable larger than the set that can be generated from Dijkstra's primitives, one might suspect that some of the advantages of the original set are lost. We believe that this is not the case. The advantages of Structured Programming seem to be based on the decomposability of the programs created and on the fact that textual progress and computational progress are related in a simple way. Both properties are maintained in our system.

1) By definition, programs are decomposable with respect to our basic control structures. The functions of these structures are clearly not any more difficult to understand than the functions of Dijkstra's primitives: Form 2a and 2b can be understood by enumerative reasoning (4) whereas

2c, i.e. the loop, can be analyzed by inductive reasoning. If formal program verification is persued, we shall need to determine one assertion for every exit of such a basic structure; whereas, for Dijkstra's primitives only one assertion is needed (since there is only one exit). On the other hand, the assertions needed for our structures will frequently be slightly simpler since we shall not need control variables that keep track of computed conditions.

2) The relation between textual and computational progress is similarly simple in both systems.

Loop free single entry graphs have a fixed and, thus, trivial relation between computational progress and textual progress.

Further, all loops are entered only at the loop head for every turn through the loop. Hence, a counter incremented when the loop head is passed, can keep track of a loop in progress. The rump of a loop is again either a loop free single entry graph or it contains other single entry loops. The number of active counters depends, as in Dijkstra's system, only on the depth of loop nestings.

We conclude that our set of primitives leads to similarly transparent programs as do Dijkstra's.

There is, however, one major difference that makes manipulating our flow graphs somewhat more difficult.

Structured Programs in Dijkstra's sense are unambiguously decomposable if one ignores the fact that the order in which concatenated actions are associated is arbitrary. This ambiguity does not matter at all because it does not cause any uncertainty since the actions are fully ordered.

For our system, this very ambiguity causes somewhat of a problem:

Because we concatenate multiple exit nodes rather than single exit nodes, the structure created by repeated concatenation is only partially ordered. Thus, the way in which the parts are put together makes a structural difference. Nevertheless, if suitable language notations are provided that permit the realization of the primitives as program text, the programmer can (and must) indicate how he wants the operations to be grouped. The programmer's wishes are lost, though, after the program is compiled. However, this fact cannot be counted to heavily against our system since the clarity of source programs and not that of object programs is the main goal.

New Control Constructs

Before introducing the new control constructs, we should note that there are at least two known constructs that support, to some extent, our system:

1) Wulf's leave feature in his programming language BLISS [12], and
2) The exit feature as available in ALGOL 68 [8].

These features are particularly effective because in both languages statement compounds may deliver values. Hence, statements may be inserted into conditional statements. The leave (exit) statement facilitates the termination (completion) of the inserted statement at any point within this statement.

Although both constructs provide quite flexible mechanisms for dealing with complex conditions, none of them fully supports the general loop structure (form 2c); further, the exit construct only permits transferring control to the next higher block level. Contrarily, the new constructs suggested fully support our system; also, they do not require that statements possess values.

Two types of control constructs are considered; the first type (A) is a generalization of existing constructs. It has been designed to facilitate the definition of arbitrary conditions within conditional statements. The

second construct (B) has been added in order to completely support form 2c.

A) As an extension of Wirth's programming language PASCAL, we suggest the notion of a group, i.e. a compound statement with potentially multiple exits. These exits are exploited by inserting the group into a conditional statement in the place of the usual expression. Therefore, a group should behave like an expression, i.e. it should possess a value. A group is formed by the brackets

begin - succeeds,

begin - fails, or

begin - group.

The first two forms are used to denote boolean groups, i.e. groups that can be used instead of boolean expressions. The last form is non-boolean and can be used in the case construct.

Examples:

(1) while begin<statement>succeeds
do<statement>

(2) case begin<statement>group of
L1: <statement>
L2: <statement>
.
.
.
end

The value of a group is determined by the statements

success,

failure, or

case<const>.

The statements success and failure are used in boolean groups, and the statement case<const> is used in non-boolean groups. These statements transfer control to the end of the group and assign the group the value true (success - succeeds, failure - fails), false (success - fails, failure - succeeds), or the value of <const>.

Example:

```
begin repeat<statement>
    until begin repeat if<cond1>then success
                until<cond2>; failure
          fails
    end
```

In the compiled program, the values defined by a group would usually not materialize; instead, each branch to the end of the group would be extended to the program part to be executed next. Since transfer of control goes to the end of a group and since the group brackets can be set arbitrarily, it is possible to specify loop terminations that bridge across several loop levels.

B) As a second structure, a new loop construct is suggested that combines the case and the while statement. Its form is

```
repeat case<expression>of
L1:  <statement>
L2:  <statement>
.
.
.
exit on
```

M1: <statement>

M2: <statement>

.
.
.

end

This statement executes the statement labeled by L1 (L2, L3, etc.) repeatedly while <expression> yields L1 (L2, L3, etc.), and it terminates the loop executing the statement labeled by M1 (M2, M3, etc.) when <expression> yields M1 (M2, M3, etc.).

Summary

The set of basic flow structures identified in this paper permits the construction of all flow graphs that do not contain multiple-entry loops. These are the same graphs as those that can be fully reduced by Allen and Cocke's method of interval reduction. It has been pointed out that programs based on the described system have most, if not all, of the positive characteristics of Structured Programs. Further, by extending Wirth's language 'PASCAL', a set of new control constructs has been suggested that support the proposed set of flow structures. We might add that PASCAL extended in this way, should not anymore contain the goto-statement.

References:

1. Allen, F. E. and Cocke, J., "Graph-theoretical constructs for program control flow analysis," IBM Research Report RC3923, July 1972.
2. Bohm and Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules", CACM. 9, 5, May 1966.
3. Dijkstra, E. W., "Goto statement considered harmful", Letter to the Editor, CACM, 11, 3, March 1968.
4. Dijkstra, E. W., "Notes on structured programming", August 1969.
5. Hopkins, M. E., "A case ~~against~~ for the GOTO", ACM Annual Conference 1972.
6. Knuth, Floyd,"Notes on avoiding 'GOTO' statements", Technical Report CS 148, Stanford University, January 1970.
7. Leavenworth, B. M., "Programming with(out) the GOTO", ACM Annual Converence 1972.
8. Lindsey, C. H. and Van Der Meulen, S. G., "Informal introduction to ALGOL 68", North-Holland Publishing Co., Amsterdam, London, 1971
9. Mills, H. "Top down programming in large systems", Debugging Techniques in Large Systems (Ed. Rustin Randall) Prentice-Hall, Englewood Cliffs, N.J. 1971.
10. Wirth, N. "Program development by stepwise refinement", CACM 14 (April 1971).
11. Wirth, N. "The programming language Pascal" Acta Informatica, 1,35-63
12. Wulf, et al. "Bliss: a language for systems programming", CACM, December 1971.
13. Wulf, W. A., "A case against the GOTO", ACM Annual Conference 1972.
14. Wegner, Eberhard., "Tree-structured programs", CACM, 16, 11, November 1973.
15. Kosaraju, S. Rao., "Analysis of structured programs", Proc. of 5th Annual ACM Symp. on Theory of Computing, May 1973.