SU-SEL 75-O 12 SU-326-P.39 – 1

The Complexity of Control Structures and Program Validation

by

Joseph W. Davison

May 1975

Technical Report No. 92

The work described herein was supported by the Energy Research and Development Agency (formally AEC) under contract AT(1 1-1)3288 and AT(04-3)326P.A. No. 39

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY · STANFORD, CALIFORNIA

SU-SEL 75-012 SU-326-P.39 - 1

THE COMPLEXITY OF CONTROL STRUCTURES AND PROGRAM VALIDATION

by

Joseph W Davison

May 1975

Technical Report No. 92

DIGITAL SYSTEMS LABORATORY

Stanford Electronics Laboratories

Stanford University

Stanford, California

The work described herein was supported by the Energy Research and Development Agency (formally AEC) under contract AT(11-1)3288 and AT(04-3)326P.A.#39

Digital Systems Laboratory
Stanford Electronics Laboratories

Technical Report No. 92

May 1975

THE COMPLEXITY OF CONTROL STRUCTURES AND PROGRAM VALIDATION

by

Joseph W Davison

ABSTRACT

A preliminary examination of the influence of control structures on the complexity of the proof of correctness of computer programs. A block structured proof technique is defined and studied. Two parameters affecting the complexity of the proof are defined; the number of exits from a block, and the cycle rank of a block, a measure of loop complexity. Proof Complexity classes of flowcharts are defined, with maximum values for these parameters. The question investigated is: How does restricting the complexity affect the class of functions realizable, assuming a given set of primitive actions and predicates: It is found that loop complexity may be traded for exits, and that for a given number of exits there are functions requiring any specific loop complexity. Further, it is shown that blocks with two exists are considerably more powerful than those with only one. In fact, for a given maximal loop complexity, there are functions that cannot be realized with one-exit blocks, but can be realized with two-exit blocks, even if the loop complexity is restricted to essentially one internal loop per block. Looking at it the other way around, the addition of a second exit to a block allows construction of flowcharts with any specified loop complexity. This result appears to be extenable to blocks with more exits, but this has not been completed.

The work is primarily of a graph theoretical nature, and may also be interpreted as an examination of sequential control structures from the point of view of feedback loop complexity.

PREFACE

With the publication of this report we are resuming our series: "Studies on Computer Organization" which, since 1970, has been supported by the Atomic Energy Commission (now Energy Research and Development Agency), first at the Johns Hopkins University and since the beginning of this year at Stanford. The attached report describes work performed at Johns Hopkins University under a predecessor contract but concluded after the termination of that contract.

Michael J. Flynn

CONTENTS

1. Introduction

1.1 Flowcharts and flowchart schemas 1

1.2 Proofs of correctness 4

1.3 Structured programming and structured programs 7

2. Literature survey

2.1 Program validation 9

2.2 Program schemas 11

3. Definitions and results

3.1 Block structured proofs 17

3.2 Loop complexity -- cycle rank of a flowchart 18

3.3 Proof Complexity (PC) based flowchart schemas 20

3.4 Reducibilities 22

3.5 Summary of results to date 23

3.6 Further research 25

References 26

Appendix I 30

The Complexity of Control Structures and Program Validation

1. Introduction

We are interested in analyzing the structural complexity of computer programs. We intend to illuminate the influence of the control structure of a program on its proof of correctness. If we succeed, we will provide a theoretical basis for the intuitive feeling, shared by many, that "structured" programs are easier to prove correct, This theory, in turn, will provide a means to judge the suitability of proposed basic control structures in simplifying program documentation and certification.

1.1 Flowcharts and flowchart schemas

In our study of the structural complexity of programs, we will represent programs by flowcharts. Flowcharts are a form of directed graphs in which the nodes represent actions and predicates, and the arcs (branches, edges) represent possible sequencing: a directed arc from A to B indicates that B is a possible direct successor of A. Nodes with only one branch leaving it represent actions, and nodes with more than one branch leaving represent Predicates,

Generally, the nodes will be labeled: any unlabeled node must have only one output branch and represents the identity transformation over the program variables. The arcs leaving a predicate node may be labeled with a condition specifying under what circumstances the indicated node is to be taken as successor, By requiring conditions at every predicate node to be mutually exclusive, we disallow non-determinism. in addition, we require

each flowchart to have at least one node with no predecessor, the entrance, and at least one node with no successor, the exit.

For the most part, we will concentrate our attention on flowchart schemas, a form of flowcharts wherein the actions and predicates are not specified, That is, they are not interpreted, but are merely identified by their labels. In schemas, the only relationship implied between the various actions and predicates is that they all operate on the program variables. If two actions or two predicates have identical labels, then the respective actions or predicates are identical, In proofs, it will sometimes be convenient to consider a flowchart rather than a schema. In these cases, an interpretation of the schema will be provided.

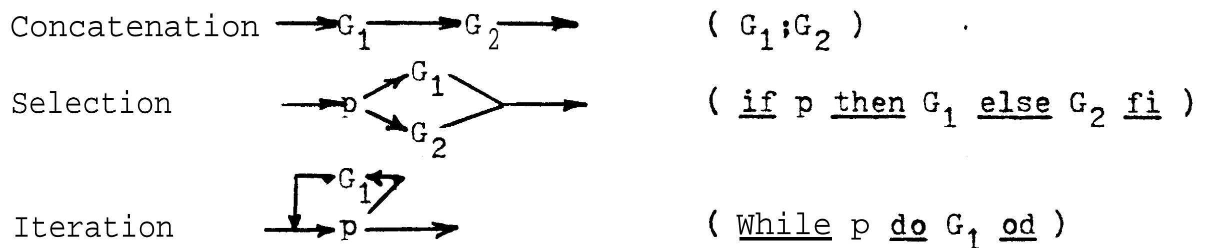
The terms control structure, control graph, flowgraph, and graph will be used synonymously with flowchart, and when the difference is not important, no distinction will be made between flowcharts and flowchart schemas.

Most of the flowcharts we will be dealing with will have a special form, called block structure, In defining a class of block structured flowcharts, we will specify a set, usually small, of "primitive blocks" and a series of rules, called block constructs, for constructing "compound blocks" from actions, predicates, and other blocks. By the general term block, we mean either primitive blocks or compound blocks constructed from some set of rules, perhaps unspecified, The primitive blocks will generally be flowcharts consisting of at most two or three actions and predicates,

Blocks will be very important in our discussions throughout this proposal, so we will define here a few of the terms we will be using. We begin with an example. The class of D-charts [5,29] is the smallest class defined byt

1. Any action is a D-chart,

2. If p is a predicate, and G_1 and G_2 are D-charts, then



are D-charts.

In this definition, we have actually given two definitions. The one on the left defined the flowchart: that on the right is intended to show another, linear representation for the same constructs, Rule 1 specifies the primitive blocks, and rule 2 recursively defines the compound blocks. In this case, the block constructs are the three subrules: concatenation, selection, and iteration. Here, each block has one entrance, one exit, and at most one internal loop. In other cases, the number of entrances and exits may be different. We therefore use the abbreviation (n_1, n_2) block to indicate that the block has n_1 entrances and n_2 exits.

This definition is a 'bottom-up' description, because it explains how to construct a D-chart from the primitives. Frequently, we will be interested in a 'top-down' description; given a flowchart, we want to see how it could be constructed from

D-charts (or other blocks, as the case may be). Thus it is necessary to "decompose" the given flowchart into blocks fitting the given description. For example, we illustrate a flowchart and one decomposition into D-charts in Figure 1 below.

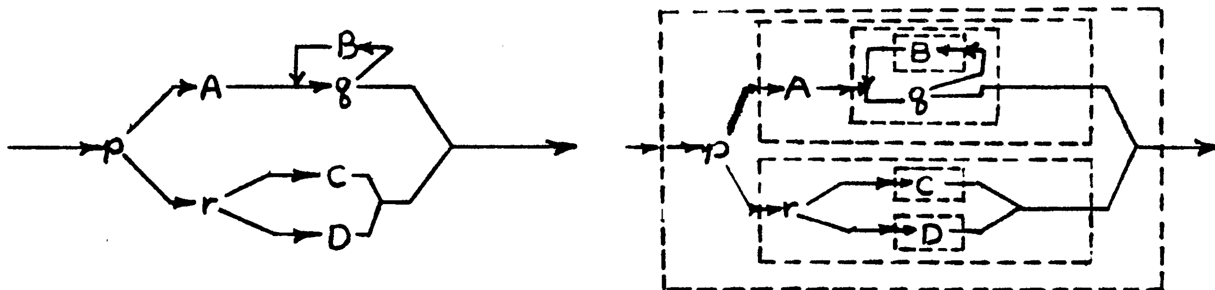


Figure 1. A flowchart and a decomposition as a D-chart.

We show the blocks by surrounding them with dashed lines; In the case of D-charts, any program has a unique decomposition into blocks, except for the concatenation operation. In general, this will not be the case. Most of the classes of flowchart schemas we will be discussing will not yield unique decompositions. Therefore, unless we specify a particular choice of blocks for a given flowchart, we mean our discussion to be true of any of the allowable decompositions.

In general, when we discuss a block, we consider any other blocks used in its construction to be non-decomposable. Thus, when we say that a D-chart block has only one internal loop, we mean it has only one loop when inner blocks are considered single nodes.

1.2 Proofs of correctness

We intend to analyze flowchart structures to illuminate their influence on proofs of correctness. There are several approaches to proofs of correctness. To name a few:

a) Structural induction [6]
b) Computational induction [26]
c) Inductive assertions [14, 25, 33]
d) Recursion induction [23, 24, 26]

As the names imply, they are all variations on the principle of mathematical induction. We are primarily interested in the inductive assertion method developed by Floyd [14], Naur [33], and Manna [25]. In this method, a logical proposition involving the variables in the program is associated with each edge in the flowchart, These propositions, called assertions, are so chosen that each is true whenever, in the course of computation, control reaches the edge tagged with that assertion. The first assertion is essentially a statement of the input data specifications, and the final assertion is a statement of the output requirements. The correctness of the program is established by showing that thesemantics of each statement, together with its antecedent assertion, logically implies its consequent assertion. Thus, the conjunction of all the implications implies that if the initial assertion is true for the input data, then the final assertion will be true on completion of execution of the program. However , this procedure proves only the partial correctness of the program. This is not total correctness, because there has been no proof that the program actually terminates, and thus the final assertion may never be reached. In general, a separate proof of termination is require;. This proof is frequently more difficult, both practically and theoretically, than the proof of partial correctness.

In practice, it is necessary for the individual constructing a proof of correctness to tag with assertions only the entrance, the exit, and a set of internal edges. The internal edges are chosen so that there are no loops in the flowchart that do not have at least one edge tagged. This divides the flowchart into a finite number of finite paths, each of which has an assertion at both ends, and includes no other tagged edges. For each such path, the programmer must verify that the antecedent assertion, together with the semantics of the statements along the path, implies the consequent assertion.

This description illustrates the relationship between the structure of a program and its proof of correctness that we seek to examine. The number of assertions that one needs to supply in order to prove correctness depends strongly on the loop structure of the program in question. For example, consider the two flowcharts in Figure 2, Each contains four predicates and two loops, but one requires two internal assertions, and one requires only one. We have not shown the action nodes in these flowcharts in order to show more simply the structure.

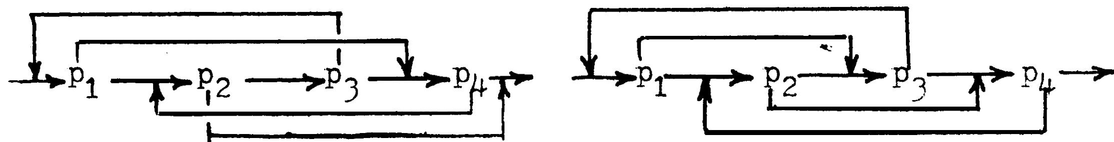


Figure 2. a) One assertion required. b) Two assertions required.

We require any measure of the complexity of these flowcharts to reflect this difference in the number of assertions required.

1.3 Structured programming and structured programs.

The goals and the significance of this research lie in the field of structured programming, Structured programming is a term used loosely to describe a discipline for the development of computer programs. The purpose of the discipline is to improve the comprehensibility of programs in order to ease she tasks of documenting, certifying and maintaining programs. "he major idea involved is the restriction of the control structures available to the programmer. The programs produced under this methodology, called structured programs, have block structured flowcharts with severely restricted primitive blocks and block constructs. Since the GOT0 statement in most programming languages represents a branch capable of connecting any two nodes in a flowhart, it is very powerful and can easily be used to construct any flowchart. For this reason, the advocates of structured programming call for its abolition, or at least for severe restrictions on its use. This has resulted in structured programs also being known as GOTO-less programs.

There is as yet no general agreement on the set of primitive blocks and block constructs to allow in structured programs. However, most proposed sets are restricted to one entrance, one exit, and at most one intern31 loop, In fact, the most controversial proposed structures, the so-called "escape" mechanisms, are precisely those that do not fall in this class.

By studying control structures to see how the complexity of woofs of correctness is affected by allowing more exits or more complex looping structures in blocks, we will be able to judge

the suitability of proposed control structures.

Our concern for the complexity of the proof of correctness does not arise because we are worried about proving programs correct. We realize that most programmers will never attempt a formal proof of correctness. On the other hand, almost all programmers try to convince themselves that their programs are correct, and anyone who must modify an existing program generally must understand the program in order to modify it successfully. These processes require at least an informal, intuitive proof of correctness, and an increase in the complexity of the formal proof probably implies a corresponding increase in the complexity of the informal proof. Norse, in the informal proof, if the complexity becomes very great at all, it will probably be impossible for the programmer ever to convince himself that the program is correct.

2. Literature Survey

The literature related to our proposed research may be grouped into two general classifications, program validation and program schemas.

2.1 Program validation

While a great deal of work has been done in this area, most of it has no bearing on our own work, and will not be discussed. As explained in the introduction, the inductive assertion method developed by Floyd[14], Naur [33], and Manna [25] motivates our study of program complexity. We illustrate the process with an example, Figure 3. This is the flowchart of a simple program to find the absolute value of the difference between its two input variables: the domain is the set of non-negative integers.

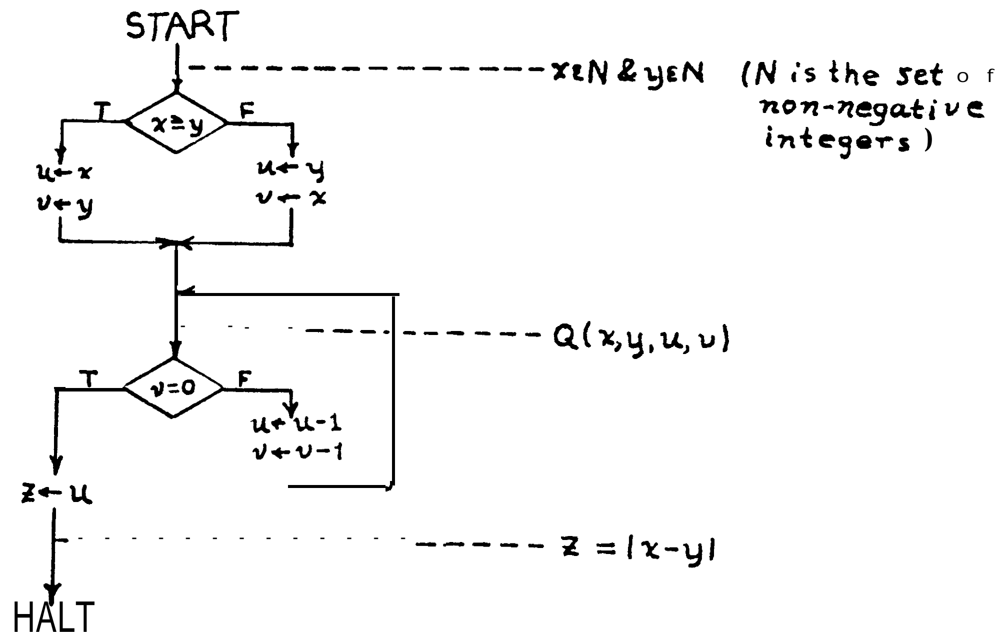


Figure 3. A flowchart with assertions.

A proof of partial correctness must show that for every non-negative integer value for x and y, the program calculates $z=|x-y|$ if it terminates. Total correctness requires, in addition, that the program terminate for all non-negative integer values for x and y. By tagging edges in the flowchart with assertions (logical propositions relating the variables) in such a way that every looping path includes at least one assertion, we divide the flowchart into a finite number of finite paths to be verified. A path is verified by showing that the antecedent assertion, together with the semantics of the programming language involved, logically implies the consequent assertion. This logical implication for a given path is called a verification condition for the path. Thus, if and only if the assertion at the entrance to the flowchart logically implies the conjunction of all the verification conditions in the program for any value of the input variables, the program is partially correct, Similarly, if we can show that there is no assignment of values to the input variables satisfying the conjunction of the entrance assertion and all the verification conditions, modified by replacing the exit assertion with its negation wherever it occurs, then we have proved total correctness. We call. the first proposition "Pc," and the second "Tc." For the flowchart of Figure 3, they are as follows, where Q(x,y,u,v) is an unspecified assertion attached as shown:

$$\text{Pc:}\quad (\forall x)(\forall y)((x\in N \;\&\; y\in N) \supset ((x\geq y \;\&\; Q(x,y,x,y)) \vee (x<y \;\&\; Q(x,y,y,x)))$$
$$\&\; (\forall u)(\forall v)(Q(x,y,u,v) \supset ((v=0 \;\&\; u=|x-y|)$$
$$\vee\; (v\neq 0 \;\&\; Q(x,y,u-1,v-1)))))$$

$$Tc: (\exists x)(\exists y)((x\in N \;\&\; y\in N) \;\&\; ((x\geq y \;\&\; Q(x,y,x,y)) \vee (x<y \;\&\; Q(x,y,y,x)) \;\&\; (\forall u)(\forall v)(Q(x,y,u,v) \supset ((v=0 \;\&\; u\neq|x-y|) \vee (v\neq 0 \;\&\; Q(x,y,u-1,v-1)))))$$

To prove partial correctness, it remains necessary to discover some assertion, $Q(x,y,u,v)$ that makes Pc true. It is not difficult to see that $Q(x,y,u,v)=(u\geq 0 \;\&\; v\geq 0 \;\&\; u=|x-y|+v)$ satisfies Pc, Proving that there is no assignment to Q that satisfies Tc is considerably more difficult. In general it may be done in one of two ways. We may prove it directly, by showing that Tc is self-contradictory for any assignment of Q, or we may prove it indirectly by proving that the program terminates from other considerations. In this case it is easiest to prove termination by noting that when control first enters the loop, $v\geq 0$, and that each time around the loop v is decremented. Since v is an integer, and it is decremented by 1 each time, it must eventually be the case that $v=0$, in which case control exits the loop and the program. Thus the program is totally correct. This approach is due to Gorn and Floyd.

One nice feature of this formulation of the proof process is that it makes apparent the fact that the most difficult part of the proof frequently is finding the right assertions.

This method is not obviously applicable to recursive programs, Others have developed methods for dealing with these more general control structures, and much work has been done in mechanizing the proof process, We will not discuss these results, however, since they are u-related to our work,

2.2 Program schemas

There are two general types of research in this area, studies of "program schemas" and studies of "flowcharts." The

studies of program schemas deal with certain decision problems concerning various classes of schemas. Examples of the types of questions examined are:

1) The termination problem -- Does a given schema halt for every interpretation?

2) The divergence problem -- Does a given schema diverge (fail to halt) for every interpretation?

3) The equivalence problem -- Do two given schemas compute the same result from the same input for all interpretations?

As might be expected, the answer to most of these extremely broad questions is undecidable, in fact, usually not even partially decidable. This means, for instance, there is no algorithm that will always halt with the correct answer for the equivalence problem when given two schemas, Manna [27] provides a concise overview of this work, This research is not directly related to our work.

On the other hand, the research concerning flowcharts is closely related to our own. This work is generally concerned with different types of equivalence or reducibility between programs or flowcharts.

The idea of block structured flowcharts is due to Böhm and Jacopini [4]. They demonstrate how to produce a D-chart that is equivalent to any given flowchart, in the sense that both, given the same input, produce the same output. In order to do this, we introduce a stack, and a test for the top element of the stack. The D-chart simulates the equivalent chart by encoding path information on the stack.

Cooper [7] shows that one can do even better, in the sense that one never needs more than one loop in the equivalent D-chart,

In this case, we associate a boolean variable with each statement in the program to be simulated, Initially the variables are set to false, except for the start statement. Then, in an outer loop, each boolean variable is tested in turn, and the associated statement is executed only if the variable is true, After executing the statement, the associated boolean variable is reset to false, and the variable for the next statement to be executed is set to true. While these results have some theoretical interest, they are of little interest in practice, since the programs they produce are even more difficult to understand than the programs they simulate.

Knuth and Floyd [18], Aschcroft and Manna [1], and Bruno and Steiglitz [5], show that there exist flowcharts that are not equivalent to any D-chart, under slightly different forms of equivalence, all of which essentially require that no new variables be added, and exactly the same sequence of actions occur in both flowcharts for a given input.

Kosaraju [20] investigates the structural complexity of the classes of block structured flowchart schemas that have been proposed as the basis for structured programs, His investigation is primarily in terms of a reducibility that requires that if X is reducible to Y, every action and predicate in Y must occur in X, and the result computed by X be the same as that computed by Y for every input value; if one does not terminate, neither does. He shows that there exists a hierarchy of classes of flowchart schemas, with D-charts at the bottom. In each class, there exist flowchart schemas that are not reducible Lo any schema in any lower class. Two of the classes he defines will be of interest to

us. These are defined below.

RE_n-charts (for Repeat-Exit) The minimal class defined by:

1. Any action is an REn-chart.
2. The (1,1) statement EXIT(i), $0 \le i \le n$, is an RE_n-chart.
3. If p is a predicate, and G_1,G are RE_n-charts, then G_1;G_2 and if p then G_1 else G_2 fi are RE_n-charts.
4. If G is an RE_n-chart, then RPT;G;END is an RE_n-chart.

(Note: We do not give a graphical form because it is difficult to describe concisely a graph structure that suits the intuitive notion of the flowchart for the RE_n-charts.) In this class, the RPT-END pair delineates a block. With this definition of a block, we can define levels in a manner analogous to that of well-formed parenthesized expressions. We number the levels so that the outermost level is level 1, and each successively deeper level increases the level number by one. When it is useful, we attach subscripts to the RPT and END statements to identify matching pairs. Thus, in Figure 4, the innermost RPT-END block (shown with subscript 3) is at level 3. Both the RPT_2-END_2 and RPT_4-END_4 pairs delineate blocks at level 2.

The RPT, EXIT(i), and END statements are purely control statements; they do not transform data in any way. The RPT statement serves only as a labeled node; when an END statement is reached, control transfers immediately to its associated RPT, Thus, the RPT-END pair represents a loop in the flowchart. The EXIT(i) statement provides a mechanism for transferring control out of a loop. It transfers control out of the ith RPT-END block enclosing the statement; the next statement executed is that statement immediately following the END statement of the outermost block exited,

```
RPT_1;
  RPT_2;
    if p then EXIT(2)
         else RPT_3;
                if q then EXIT(2)
                     else if r then EXIT(1)
                                else A
                     fi
                fi;
              END_3
    fi;
  END_2;
  C;
  RPT_4;
    if s then EXIT(2)
         else B
    fi;
  END_4;
END_1
```

Figure 4. An example of an RE_2-chart.

Thus, in Figure 4, the EXIT(2) statement in the innermost block (with subscript 3) transfers control to the statement "C" following the END_2 statement, and the one in the RPT_4-END_4 block transfers control out of the program, If we consider some RPT-END block that contains another RPT-END block nested k levels deeper, and this nested block contains an EXIT(i) statement at its top level, then if $i \geq k$, that exit statement has an <u>effective level</u>

of i-k with respect to the outer block. No outermost block of an RE_n-chart contains an exit with an effective level>0.

GRE_n-charts (Generalized RE_n):

The class of GRE_n-charts is defined the same as RE_n-charts, except we do not restrict i in an EXIT(i) statement to be less than n. Instead, we restrict each RPT-END block to have at most n distinct effective levels.

Peterson, Kasami, and Tokura [34] consider a stronger form of reducibility, called size reducibility. $X \leq_s Y$ if and only if for every input, X produces exactly the same sequence of actions and predicates as Y, and the total number of actions and predicates in X (the size of X) is no less than the size of Y. The results are largely subsumed by Kosaraju's, in that they show the existence of programs not size- reducible to D-charts, programs not size reducible to RE_1-charts, and programs not size reducible to $\bigcup_{i=1}^{\infty} RE_i$-charts. We abbreviate this last class as RE_∞. Finally, an algorithm is given that produces, for any given flowchart, an RE_∞-chart to which it is reducible in the same sense as above, except that the size restriction is dropped.

3. Definitions and Results.

3.1 Block structured proofs.

In order to study the effects of the structure of flowcharts on proof complexity, it is necessary that we have a particular form of proof in mind. We are particularly interested in proofs of **structured programs" , or, equivalently, of block structured flowchart schemas. The inductive assertion technique, as previously discussed, does not take into account the block structure of a given flowchart: rather, it attempts to handle the entire flowchart at one time.

We extend the idea of verifying paths to verifying blocks. For each innermost block, we provide assertions for each entrance, and for each exit, and then verify that the block satisfies these conditions by considering its internal structure. Once this has been done, we may ignore the structure of the internal blocks in verifying the correctness of any block that is constructed from actions, predicates, and previously verified blocks.

Our goal in proposing such a block structured proof process is to reduce the complexity of the task of proving a program correct. When confronted with the task of proving the correctness of almost any reasonable program, we may be dismayed by the magnitude of the problem until we begin to form opinions of what function is served by the various parts of the program, It is this idea that we try to capture in our block structured proofs. We reduce the overall complexity of the single task by dividing it into a number of smaller tasks. As Cooper [9] observes:

> "In some ways it seems natural to isolate a part of the program and specify its properties, then

combine these properties to obtain the conditions to be verified. Thus one is led to consider block form, attaching relations (between values of variables on input and output) to blocks and deriving conditions between these relations depending on the particular block structure,"

Given this idea of a block structured proof, it is not difficult to apply it in a "top-down" manner, as advocated by Mills [29-32], Dijkstra[11,12], Wirth [35], and others.

We are not interested in developing a particular theory of block structured proofs. We feel that such proofs can be generated by rewriting a given Floyd/Naur proof, and, for our purposes, this observation will suffice. We are interested, in fact, in only one aspect of these proofs: The required points of attachment of the assertions. These points are: All entrances to the block; all exits from the block; and one in each loop within the block. We are guided in our choice of block constructs by a need to relate these assertion points to the structures.

3.2 Loop complexity -- cycle rank of a flowchart

Before we can describe any particular block constructs, we need to examine more closely the idea of loop complexity of a flowchart, and the number of internal assertions (those not on entrances or exits) required. We consider assertions attached to nodes rather than to edges; we assume the attached assertion is true at the entrance to that node no matter how it is reached. In fact, normally we consider tagging only predicates. This is possible in general only when we restrict our attention to "proper" flowcharts, flowcharts in which there exists, for each node, a "consistent path" from some entrance to some exit of the flowchart which passes through that node. Here, by consistent path we mean

one on which there are no inherently contradictory predicates, This restriction to proper flowcharts prevents, for instance, loops that contain no tests.

We define the cycle rank of a flowchart to be the minimum number of assertions required to insure at least one in each loop. This definition is stated more precisely below.

Let F be a flowchart, X be a set of nodes, each in some loop in F, and G be the graph obtained from F by removing all nodes in X. Then, if G is loop free, any node in X is called a breaknode. The cycle rank of F is the number of nodes in a minimal set of breaknodes for F.

This is not really sufficient to characterize the loop complexity of a graph, as may be seen by examining the graphs in Figure 5. Each has cycle rank 2, even though the complexity of the loop structure seems to vary widely. We will remedy this problem in the next section.

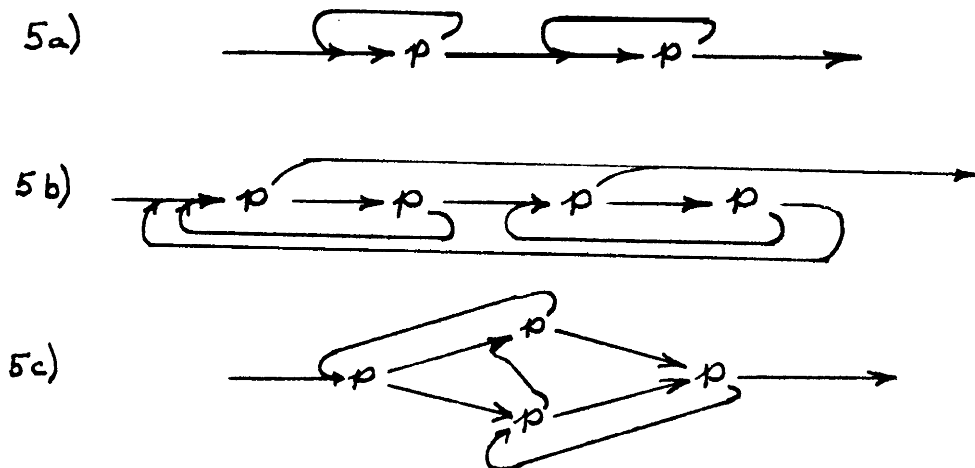


Figure 5. Flowgraphs with cycle rank 2 (action nodes omitted)

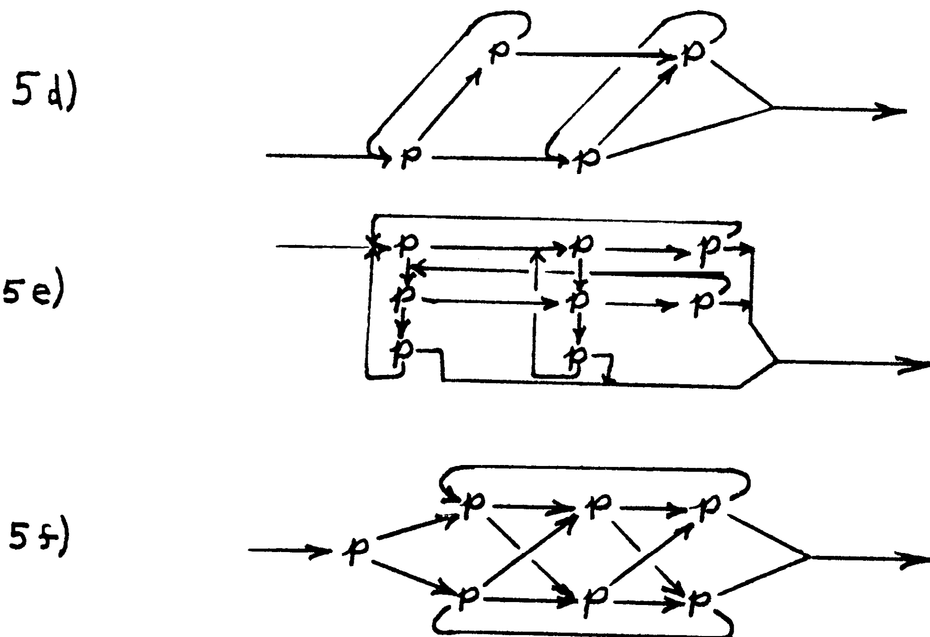


Figure 5. [continued)

3.3 Proof complexity (PC) based flowchart schemas

In the block structured proofs previously discussed, we saw that the number of assertions required to prove a block correct depends on the number of entrances, number of exits, and the cycle rank of the block. We therefore define a series of classes of flowcharts with these items as parameters. We call these classes proof complexity classes, and denote each of them as $(i,k)PC_r$-charts. Here, i,k, and r are integers, interpreted as follows:

- **i:** The maximum number of entrances to a block.
- k: The maximum number of exits from a block.
- **r:** The maximum cycle rank of a block construct.

By cycle rank of a block construct, we mean the rank of the construct with any component blocks considered as single nodes. The fact that this cycle rank is restricted to the block constructs

causes no problem, since our recursive definition (below) insures that every component block also has cycle rank no greater than r.

$(i,k)PC_r$-charts: The smallest class of flowcharts defined by:

1. Any (j,h) flowchart, $0 \leq j \leq i$, $0 \leq h \leq k$, composed of actions and predicates with cycle rank no greater than r is an $(i,k)PC_r$-chart.
2. Any (j,h) flowchart, $0 \leq j \leq i$, $0 \leq h \leq k$, composed of actions, predicates, and $(i,k)PC_r$-charts with block construct cycle rank no greater than r is an $(i,k)PC_r$-chart.

We reserve the term $(i,k)PC_r$-chart to refer to a whole graph in the remainder of this report, and use the terms "block" and "$(i,k)PC_r$-block" to mean an $(i,k)PC_r$-chart either used as a component or as a whole flowchart. If a flowchart belongs to some class of $(i,k)PC_r$-charts, it belongs to all classes with numerically greater parameters. In addition, by choosing different types of blocks, increasing the number of entrances and/or exits, it is often possible to choose smaller blocks, yielding lower cycle ranks. For example, we may classify the flowcharts of Figure 5 as follows:

5a) $(1,1)PC_1$
5b) $(1,1)PC_2$ and $(1,2)PC_1$
5c) $(1,1)PC_2$ and $(2,2)PC_1$
5d) $(1,1)PC_2$ and $(2,2)PC_1$
5e) $(1,1)PC_2$ and $(2,3)PC_1$
5f) $(1,1)PC_2$ and $(3,3)PC_1$

Note that this multiple classification differentiates these flowcharts into five different groups. We feel that these groups more closely reflect the complexity of the looping constructs in these figures than did the simple assignment of cycle rank 2 to all of them.

3.4 Reducibilities

We intend to investigate the structural complexity of (i,k)-PC_r-charts using two different reducibility orderings. We will follow much the same approach as Kosaraju [20], (he studied a different family of classes of schemas) in his research. We define a flowchart schema, X, to be elementally reducible to a flowchart schema, Y, ($X \subseteq_E Y$) if and only if every element (action or predicate) of Y is an element of X, and, for every interpretation and for every input, u, X and Y both either terminate with the same value, or fail to terminate. Intuitively, X and Y both compute the same (partial) function, and the elements used in constructing Y are a subset of the elements in X. Note that we allow in Y any number of copies of any element in X, but no new elements may be added.

We call the sequence of actions and predicates encountered in a flowchart, X, while processing input u the computational sequence in X for u, written C(X,u).

A flowchart, X, is oath reducible to a flowchart, Y, ($X \subseteq_P Y$), if and only if $X \subseteq_E Y$, and C(Y,u) is a subsequence of C(X,u), for every input, u.

These orderings, as defined, relate flowcharts or flowchart schemas, but we frequently use them to relate classes of schemas. In these cases, when we write $\mathcal{X} \subseteq \mathcal{Y}$, and $\mathcal{X}$ and $\mathcal{Y}$ are classes of schemas, we mean that:

$$(\forall X \in \mathcal{X})(\exists Y \in \mathcal{Y})(X \subseteq Y) .$$

We denote that both $X \subseteq Y$ and $Y \subseteq X$ hold by $X \equiv Y$ (X is equivalent to Y), and that $X \subseteq Y$ holds, but $X \equiv Y$ does not by $X \sqsubset Y$ (X is strictly reducible to Y).

3.5 Summary of Results to date

In our investigations so far, we have been trying to understand the partial ordering of the $(1,k)PC_r$-chart classes under elemental reducibility, and also to relate these classes to Kosaraju's RE_n- and GRE_n-charts.

We lose nothing by restricting our attention to one input blocks, since given any $(i,k)PC_r$-chart, we can construct a path equivalent $(1,k+r)PC_1$ chart. We omit the details here, since they are rather involved.

In order to be able to prove any statements relating $(1,k)$-PC_r-charts, it has been necessary to develop a normal form for the blocks. This normal form is illustrated in Figure 6. Each of the sub-blocks, A and B, in the normal form are acyclic: all looping arcs are shown explicitly. The nodes $i_1 \ldots i_r$ in B are considered to be the breaknodes in this form.

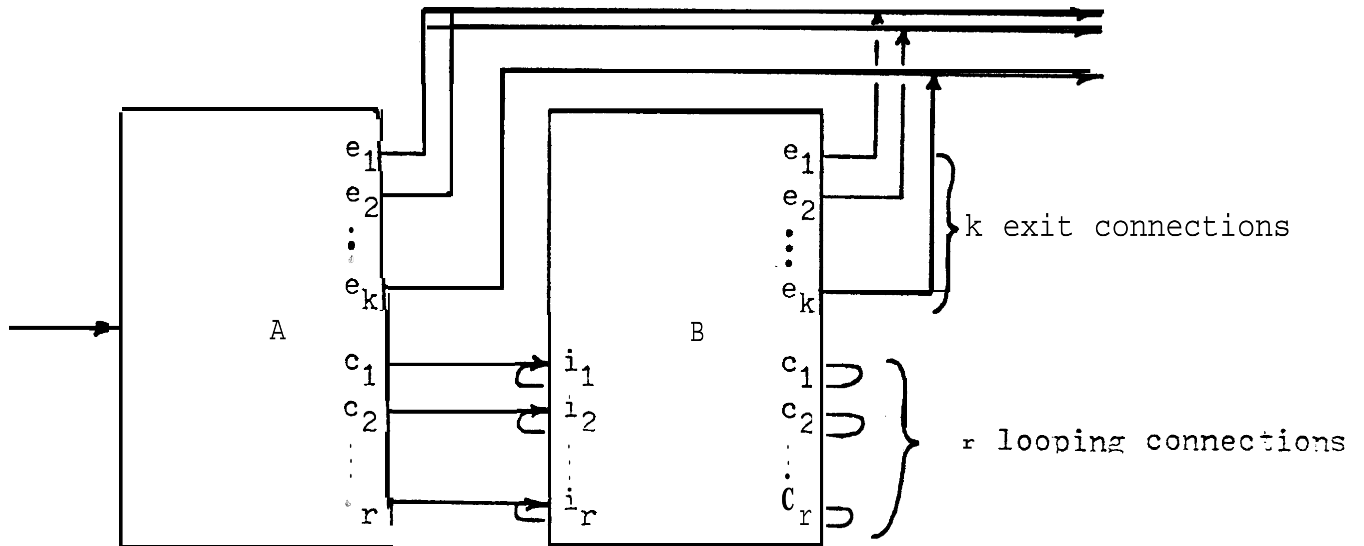


Figure 6. Normal form for $(1,k)PC_r$ blocks. A and B are acyclic.

We can easily show that every $(1,k)PC_r$ block has a path equivalent

normal form block for any choice of breaknodes.

So far, we have been able -to prove that for any cycle rank, r, there exists a $(1,2)PC_1$-chart that is not elementally reducible to any $(1,1)PC_r$-chart, but that is a $(1,1)PC_{r+1}$-chart. 'This proof shows both that $(1,1)PC_1 \sqsubset_E (1,2)PC_1$ and that, for all r, $(1,1)PC_r \sqsubset_E (1,1)PC_{r+1}$. This strongly suggests that a hierarchy exists both for k, the number of exits, and for r, the cycle rank of the blocks; but we have not yet proved this more general result.(see the appendix for this and other proofs).

We have also been able to show that it is possible to trade cycle rank for exits, in that for any $(1,k)PC_{r+1}$ block, X, we can construct a $(1,k+1)PC_r$ block, Y, such that $X \sqsubseteq_P Y$. This can be used to show that $(1,k)PC_r$-charts $\sqsubseteq$ $(1,k+r-1)PC_1$-charts.

. . It is not difficult to show that RE1-charts are the same class as the $(1,1)PC_1$-charts. This class includes almost all the block constructs proposed for structured programming. The fact that the $(1,1)PC_1$-charts are the least complex charts in our family of schemas agrees with the intuition that structured programs are easier to prove correct than are general flowcharts.

In our proof that for any r there exists a $(1,2)PC_1$-chart that is not $(1,1)PC_r$, the flowchart used is an RE_2-chart. This shows that there are RE_2-charts that require at least r+1 internal assertions for any r. The significance of this is that the escape mechanisms in some programming languages (such as the "ON ERROR GOTO" statement in BASIC-PLUS, or the "ON CONDITION" statement in PL/1) are at least as powerful as the RE_2-charts, and may thus be used in a manner that greatly complicates a proof of correctness for the program.

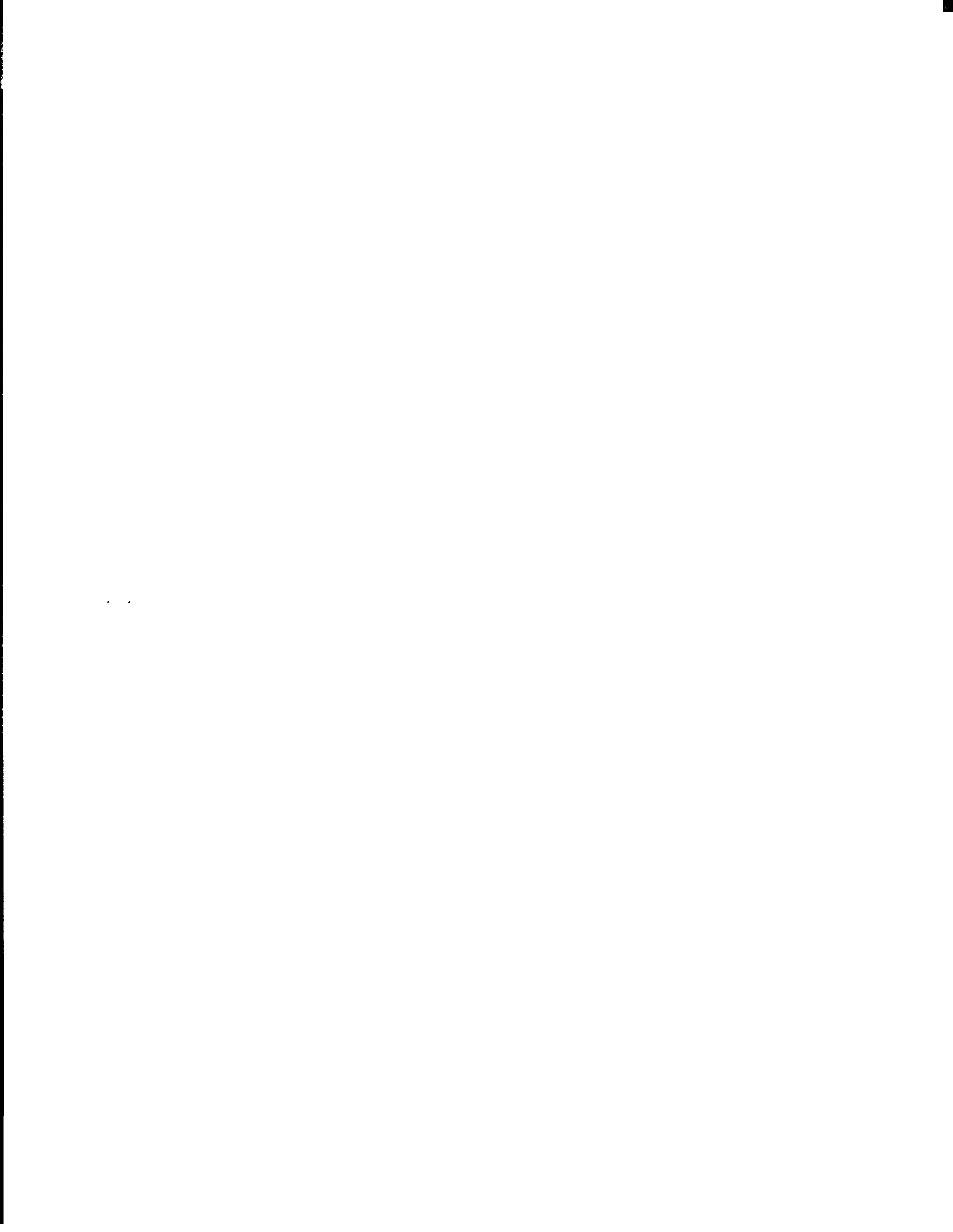
Because of the naturee of the restrictions on RE_n-charts, we do not expect to be able to show a precise relation between them and our $(1,k)PC_r$-charts. This is not true for the GRE_n-charts, however, and we expect to be able to show (we have not yet done so) that $GRE_n \equiv_p (1,n)PC_1$. So far, we can show that $GRE_n \subseteq_p (1,n)PC_1$. Intuitively, the proof is simple; each GRE_n-block has at most n distinct effective levels, and the RPT node may be taken as the only breaknode. Thus, we may consider any GRE_n-block to be a $(1,n)PC_1$-block.

It remains to show the converse.

3.6 Further research

In addition to the remaining research mentioned in the last section, we intend to survey the control structures available in t-he major programming languages in use today, and to relate these control structures to the hypothesized hierarchy. We will pay particular attention to those features that may be used to generate structures more powerful than the $(1,1)PC_1$-charts. Examples of this type of statements are the current implementations of CALL and RETURN statements in most languages.

We may then be able to suggest restrictions on the use of these statements to reduce their impact on proof complexity. In addition, we may be able to suggest additional control structures that could be added to programming languages without severely affecting the complexity of proofs of correctness.



References

1. Ashcroft, E.A. and Manna, Z., The Translation of 'GOTO' Programs to While' Programs, Report No. Stan-CS-71-188, Stanford University.

2. Basu, S.K., Transformations on Directed Graphs, Journal of Combinatorial Theory, vol. 9, 1970, pp.244-256.

3. Bochman, G.V., Multiple Exits from a Loop without the GOTO, CACM 16, July 1973, pp. 443-444.

4. Rohm, C, and Jacopini, G., Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules, CACM 9, No. 5, May 1966, pp. 366-371.

5. Bruno, J. and Steiglitz, K., The Expression of Algorithms by Charts, JACM Vol. 19, No. 3, July 1972, pp. 517-525.

6. Burstall, R.M., Proving Properties of Programs by Structural Induction, Computer Journal, Vol. 12, No. 1, February 1969, pp 41-48.

7. Cooper, D.C., Bohm and Jacopini's Reduction of Flowcharts, (letter to the editor) CACM 11, No. 8, August 1967, p. 463.

8. Cooper, D.C., Some Transformations and Standard Forms of Graphs with Applications to Computer Programs, Machine Intelligence 2, (eds. Dale, E. and Michie, D.), New York: American Elsevier, 1968, pp. 21-32.

9. Cooper, D.C., Programs for Mechanical Program Verification, Machine Intelligence 6, 1971, pp. 43-59.

10. Dijkstra, E.W., GOTO Statement Considered Harmful, (letter to the editor) CACM 11, No. 3, March 1968, p. 147.

11. Dijkstra, E.W., Structured Programming, Software Engineering Techniques, (eds. Buxton, J.N. and Randall, B.), Rome: NATO Science Committee, 1970, pp. 84-87.

12. Dijkstra, E.W., Notes on Structured Programming, Structured Programming (eds. Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R.) New York: Academic Press, 1972, pp. 1-82.

13. Elspas, B., Levitt, K.N., Waldinger, R.J. and Waksman, A., An Assessment of Techniques for Proving Program Correctness, Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147.

14. Floyd, R.W., Assigning Meanings to Programs, Proceedings of the Symposium in Applied Mathematics, Vol. 19, Providence, R.I.: American Mathematical Society, 1967, pp. 19-32.

15. Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM 12, No. 10, October 1969, pp. 576-580.

16. Kaplan, D.M., Proving Things about Programs, Proceedings of 4th Annual Conference on Information Science and Systems, Princeton, 1970, pp. 244-251.

17. King, J.C., Proving Programs to Be Correct, IEEE Transactions on Computers, Vol. C-20, No. 11, November 1971, pp. 1331-1336.

18. Knuth, D.E. and Floyd, R.W., Notes on Avoiding GOTO Statements, Report No. Stan-CS-70-148, Stanford University.

19. Knuth, D.E., Structured Programming with GOTO Statements, Report No. Stan-CS-74-416.

20. Kosaraju, R.S., Analysis of Structured Programs, Proceedings of 5th ACM Symposium on Theory of Computing, May, 1973 (to appear in Journal of Computer and Systems Sciences).

21. Linden, T.A., A Summary of Progress toward Proving Program Correctness, Proceedings of Fall Joint Computer Conference, 1972, CD. 201-211.

22. London, R.L., Bibliography on Proving the Correctness of Computer Programs, Machine Intelligence 5 , (eds, Meltzer, B. and Michie, D.), 1970, pp. 569-580.

23. McCarthy, J., Towards a Mathematical Science of Computation, Proceedings of IFIP Congress 62, (ed. Popplewell, C.M.), [illegible] , Amsterdam: North Holland Publishing Co., 1963, pp.21-28.

24. McCarthy, J., A Basis for a Mathematical Theory of Computation, Computer Programming and Formal Systems, (eds. Braffort and Hirschberg), Amsterdam: North Holland Publishing co., 1963, pp. 33-70.

25. Manna, Zohar, The Correctness of Program, JCSS, Vol. 3, 1969, pp. 119-127.

26. Manna, Z., Ness, S. and Vuillemin, Inductive Methods for Proving Properties of Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972, pp. 27-50.

27. Manna, Z., Program Schemas, Currents in the Theory of Comouting (ed, Aho, A.V.), Englewood Cliffs, N.J.: Prentice-Hall, 1973.

28. Martin, J.J., Generalized Structured Programming, VPI and SU Computer Science Department Report (undated).

29. Mills, H.D., Mathematical Foundations for Structured Programming, . Report IBM-FSC-72-6012, IBM Federal Systems Division,

30. Mills, H.D., Top Down Programming in Large Systems, Debugging Techniques in Large Systems, (ed. Rustin, R.) Prentice Hall, 1971, pp. 41-55.

31. Mills, H.D., How to Write Correct Programs and Knox It, Report IBM-FSC-73-5008, IBM Federal Systems Division, 1973.

32. Mills, H.D., The New Math of Computer Programming (in press).

33. Naur, P., Proof of Algorithms by General Snapshots, BIT 6, 1966, pp. 310-316.

34. Peterson, W.W., Kasami, T. and Tokura, N., On the Capabilities of While, Repeat, and Exit Statements, CACM 16, No. 8, August 1973, p. 503.

35. Wirth , Niklaus, Program Development by Stepwise Refinement, CACM 4, No. 4, April 1971, pp. 221-227.

36. Wulf, W.A., Russell, D.B. and Haberman, A.N., BLISS: A Language for Systems Programming, CACM 14, No. 12, December 1971, pp. 780-790.

37. Yelowitz, L., A Symmetric, Top-Down Structured Approach to Computer Program/Proof Development. Report IBM-FSC-73-5001, IBM Federal Systems Division, 1973.

APPENDIX I: The Proofs

In this appendix, we include the theorems alluded to in the text, and their proofs.

In discussing graph transformations, we make reference to "node splitting", and "node stretching". By node splitting we mean the process of duplicating a node and its output branches, and the distribution of its input branches in such a way that each copy has at least one input branch, and no input branch is duplicated. By node stretching we mean the introduction of a new node, connected to the original by a single output branch, and the distribution of the inputs as in node splitting. The new node is considered to be an identity transformation. These transformations obviously preserve path reducibility, at least when identity transformations are ignored. Figure I1, below, illustrates the use of both node splitting and node stretching in transforming more general flowcharts into D-charts,

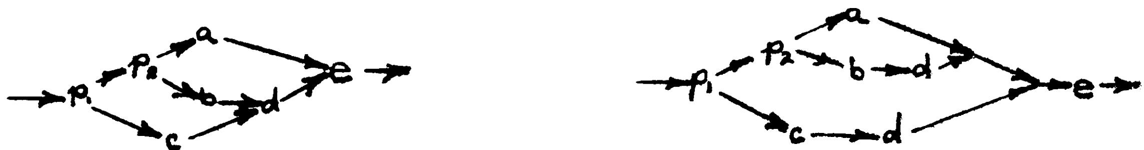


FIGURE I1: Node splitting and node stretching: node d was split, node e was stretched (twice).

THEOREM 1: For every $(1,k)PC_r$-chart, there exists a path equivalent $(1,k)PC_r$-chart in normal form.

Proof: Consider any $(1,k)PC_r$-chart, F. Assume F is not in normal form, else we are through. We describe the derivation of a path equivalent $(1,k)PC_r$-chart, in normal form, from F. Figure I2 illustrates the process for a particular case: the general procedure is described below.

First: Using node splitting as needed, copy F, but separate the paths from the entrance to the breaknodes and/or outputs from the other paths (see figure I2a and I2b).

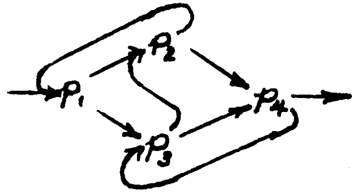
Secomdr Stretch each breaknode, B_j, $1 \leq j \leq r$, and split the introduced identity node into two nodes, c_j and c_j'. The inputs to B_j are distributed between c_j and c_j' so that all paths from the entrance to B_j include c_j, and all paths from any breaknode to B_j include c_j' but not c_j (see figure I2c).

Thirdr The normal form graph is, at most, a different arrangement of the graph produced so far. The first part consists of the paths from the entrance to one of the c_j, $1 \leq j \leq r$, or to one of the outputs. The second part, the <u>loop block</u>, consists of the paths from B_j, $1 \leq i \leq r$, to c_j', $1 \leq j \leq r$, or to one of the outputs. As shown in figure 6, page 23, c_j and c_j' (the c_j in the loop block) are connected to i_j, the j th entrance to the second block, which, in this proof, is the node corresponding to B_j.

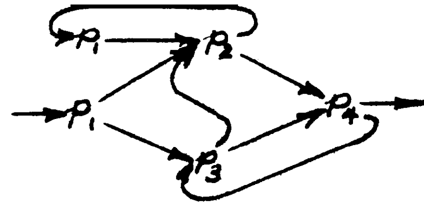
Since the only transformations involved in producing the new flowchart are node splitting and node stretching, the new flowchart is path equivalent to F. Since no new outputs or breaknodes were introduced, and the breaknodes were not duplicated, the new chart must remain $(1,k)PC_r$. The new chart is thus the required $(1,k)PC_r$-chart in normal form.

By <u>unwinding a $(1,k)PC_1$ block n turns</u> is meant the process of transforming the block to normal form, duplicating the loop block n times (resulting in n+l copies), and then connecting the loop exits from one block to the inputs of the next, the final loop exits to the initial inputs, and the outputs to each other, as

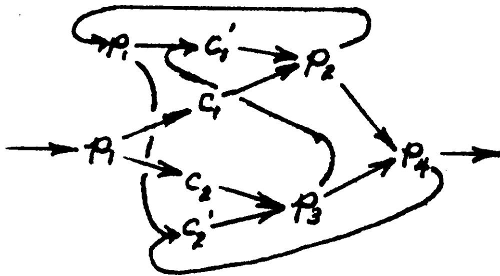
shown in figure 13. The resulting graph is obviously path equivalent to the original.



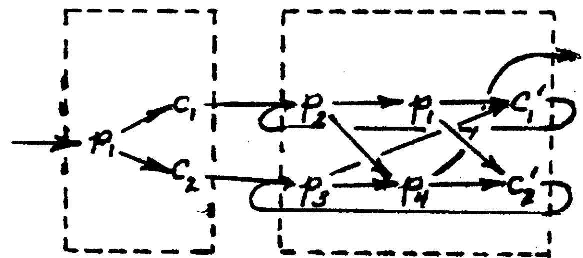
a) $(1,2)PC_1$-chart not in normal form.



b) Same graph after first step in the transformation.



c) After second step in the transformation.



d) The $(1,2)PC_1$-chart in normal form.

FIGURE 12: Transforming a $(1,2)PC_1$-chart to normal form (nodes p_2 and p_3 are used as breaknodes).

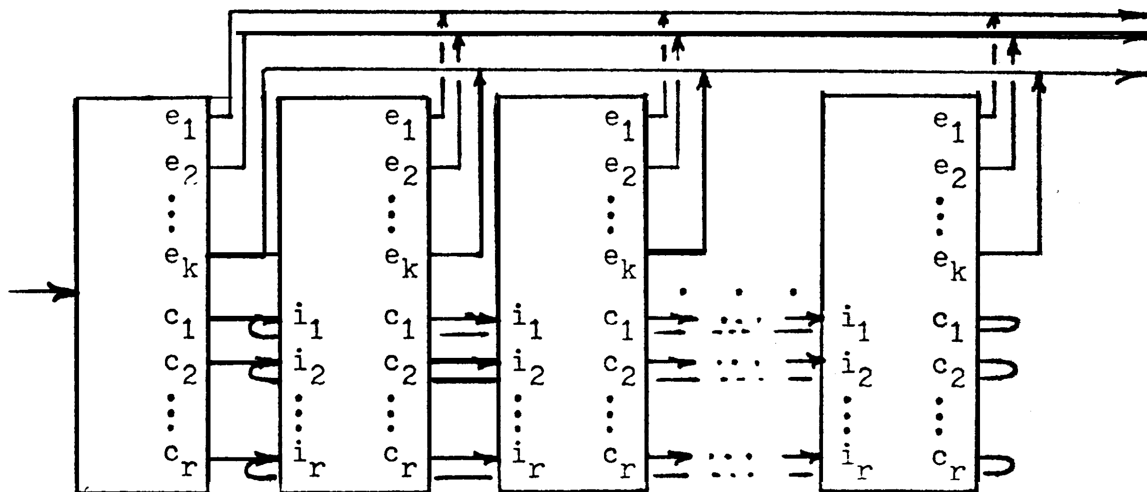


FIGURE 13: Unwinding a $(1,k)PC_r$-block.

THEOREM 2: For every $k \geq 1$, and every $r \geq 1$, for every $(1,k)PC_{r+1}$-chart, F, there exists a $(1,k+1)PC_r$-chart, G, such that $F \subseteq_p G$. That is, we may trade cycle rank for outputs.

Proof: Transform F to normal form, if need be. We construct G from F, being careful to preserve path reducibility.

The loop block of F is a (r+1,r+1+k) acyclic graph, where we consider the cycle connections to be external to the loop block. Let B be the $(r+1,k)PC_{r+1}$-chart consisting of the loop block from F, together with the cyclic paths from c_j to i_j, $1 \leq j \leq r+1$. Construct a set of r+1 $(1,k+1)PC_r$-charts, H_i, $1 \leq i \leq r+1$, from B as follows:

the input to H_j is i_j in B; the exits from H_j are e_1 through e_k, as in B, plus e_{k+1}, which is the node labeled c_{j+1} (or c_1 if $j=r+1$). The body of H_j is identical to B, except that the branch connecting c_{j+1} to i_{j+1} is deleted.

For example, see figure I4.

Next, considering each of the Hi as $(1,k+1)$ predicates, construct r+1 $(1,k)PC_1$-charts, M_j, $1 \leq j \leq r+1$, as follows:

The entrance to M_j connects directly to the input of H_j. Connect e_{k+1} in H_i to the input of H_{i+1} (or to the input of H_1 if $i=r+1$). Connect the e_1, ... , e_k exits of all H_i $1 \leq i \leq r+1$, to the corresponding output of M_j.

For example, figure I4 illustrates M_2.

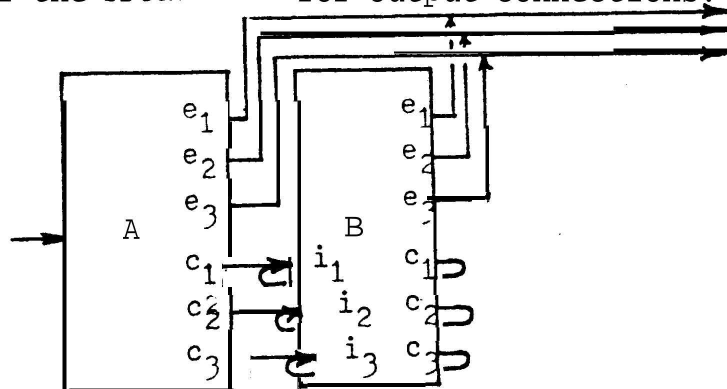
Now, at this level, each Mj appears to be $(1,k)PC_1$, but since it is composed of $(1,k+1)PC_r$-charts, it is also $(1,k+1)PC_r$.

We now produce the $(1,k+1)PC_r$-chart, G, from the M_j, and the initial block of the normal form of F. Note that the G so produced is not in normal form. Connect the c_i exit of the initial block to the input of M_i, for each value of i, $1 \leq i \leq r+1$, and connect all the exits labeled e_j to the e_j output of G, for all j, $1 \leq j \leq k$.

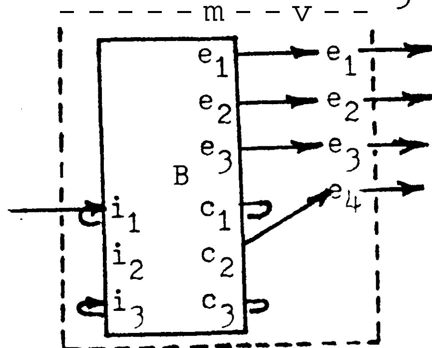
The original chart, F, is obviously path reducible to G, since the initial blocks of both graphs are identical, and M_j is so constructed that any path in F which takes control from i_j in the loop block of F to any exit is duplicated in M_j. No new paths are introduced, since each component, H_i, duplicates the loop block from F, except for the loop through i_{i+1}, and any path that exits H_i without exiting G is a duplicate of some path from i_i to i_{i+1} in the 100-p block,

COROLLARY 2.1: $(1,k)PC_{r+1} \equiv_p (1,k+r)PC_1$.

Proof: By repeated application of the theorem, we may trade all but one of the breaknodes for output connections.-

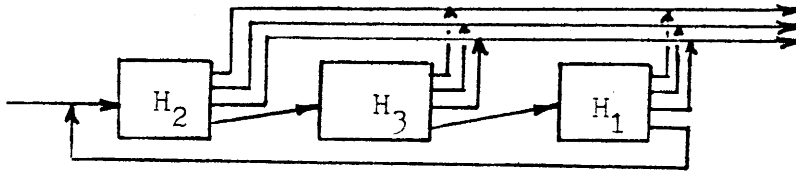


a) F in normal form: $(1,3)PC_3$.

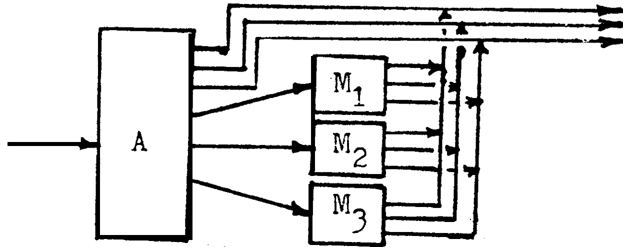


b) H_1 a $(1,4)PC_2$-chart used in an intermediate step.

FIGURE I4: Examples of the construction in Theorem 2. Here, k=3, r=2.
(continued next page)

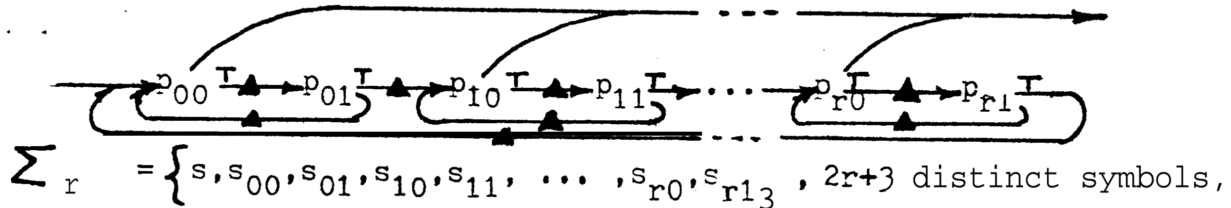


c) M_2, a $(1,4)PC_2$-chart used in an intermediate step (each of the Hi is a $(1,4)PC_2$ block.



d) G, the final $(1,4)PC_2$-chart; $F \subseteq_p G$.

FIGURE I4: Examples of the construction in theorem 2, continued.



$\Sigma_r = \{s, s_{00}, s_{01}, s_{10}, s_{11}, \ldots, s_{r0}, s_{r1}\}$, 2r+3 distinct symbols,

p_{ij} = "the prefix symbol is s_{ij}."

▲ = "delete the prefix symbol."

Inputs to F_r are strings in $^*_r$

$\mathcal{L}(F) = \{x \varepsilon \Sigma^*_r \mid$ control reaches the exit from F_r when x is input.$\}$

FIGURE I5: Phe flowchart for theorem 3.

THEOREM 3: For any given cycle rank, $r \geq 1$, there exists a $(1,2)PC_1$-chart which is not elementally reducible to any $(1,1)PC_r$-chart, but which is $(1,1)PC_{r+1}$.

Proof: Consider the flowchart, F_r, shown in figure I5, with the indicated interpretation. Note that F_r is both $(1,2)PC_1$ and $(1,1)PC_{r+1}$.

Assume, contrary to the theorem, that some (1,1)PC_r-chart, G, exists such that $F_r \subseteq_E G$. We may assume that there is at least one innermost block, π, in G, and an input string, $x_0x_1x_2$, with the following properties:

a) after deleting the final symbol in x_0, control is within π.

b) the length of x_1 is at least 5, and control remains within π at least until x_1 is deleted,

c) neither x_0 nor x_0x_1 is accepted by F_r, but $x_0x_1x_2$ is,

We may assume this without loss of generality, since, if there is a bound, uniform over all inputs, on the number of symbols deleted with control completely within any innermost block, we may unwind that block the appropriate number of times and then delete the looping branches. Thus, we may replace any such innermost block with an acyclic graph. It is not possible that all innermost blocks will be deleted by this technique, since there are strings of arbitrary length accepted by G, distinguishable from strings rejected by G only in the final symbol; this requires that G contain at least one loop, and we may choose some such loop as an innermost block.

Let σ be the prefix symbol of x_1. Then either x_0s or $x_0\sigma s$ is accepted by F_r, and thus by G, and must therefore cause control to exit π. Let x_0' represent either x_0 or $x_0\sigma$, as needed so that $x_0's$ is accepted by G. Note that control must be inside π after deleting x_0', since it is a prefix string of x_0x_1. Furthermore, after deleting x_0', control in F_r is at the entrance to some test, p_{j0}, for some value of j. Thus, every string $x_0'w$ with $w \neq s_{j0}$ is accepted by G, and therefore must cause control to exit π. Since the length of x_0x_1 is greater than that of $x_0'w$, and control remains within π until after x_0x_1 is deleted, control cannot leave π for the string $x_0's_{j0}$

($x_0 x_1$ must be an extension of $x_0' s_{j0}$).

Furthermore, control cannot exit π for the string $x_0' s_{j0} s_{j1}$, because there is only one exit from π, and from that exit there is no way to distinguish between $x_0' s_{j1}$ and $x_0' s_{j0} s_{j1}$, since no information is available there about the symbols preceding the s_{j1}. Thus, if they both exited, both would be accepted-since the former must be, but the latter is not accepted by F_r.

Again, each string $x_0' s_{j0} s_{j1} w$, $w \neq s_{(j+1)0}$, must be accepted by G, and therefore must cause control to exit π. This includes the string $x_0' s_{j0} s_{j1} s_{j0}$. Thus, since $x_0' w$, $w \neq s_{j0}$, and $x_0' s_{j0} s_{j1} s_{j0}$ cause control to leave π on their final symbol and be accepted, every string that causes control to exit π must be accepted by G.

Now,. we have , for every symbol in Σ_r, exhibited a string accepted by G that causes control to exit π on that symbol, and thus every string with prefix x_0 that causes control to exit π must be accepted by G.

Consider the string, X, shown on the next line:

$$x_0 (s_{j0} s)^a s_{j0} s_{j1} (s_{(j+1)0} s)^a \ldots (s_{r0} s)^a s_{r0} s_{r1} (s_{00} s)^a \ldots (s_{(j-1)0} s)^a s$$

This string must be accepted by G for any value of $a \geq 0$, but no prefix string of it may be, Let there be at most A predicates in π. Then the string X', being the string X with a=A, must be accepted by G, and therefore causes control to exit π. Since no prefix of X' is accepted by G, control cannot exit π except on the final symbol, s. Since there are only A predicates in π, and there are 2A symbols in each forcing substring, $(s_{i0} s)^A$, each such substring forces control to loop in π. There are r+1 such forcing substrings in X'. By definition of $(1,1)PC_r$ blocks, there are at most r breaknodes in π.

Therefore, at least two of these loops must share a common breaknode, q_π. Assume, without loss of generality, that the two forcing substrings in question are $(s_{h0}s)^A$ and $(s_{k0}s)^A$, with $h<k$. Then, for some values of m,n both $\leq A$, there exists a string of the form $Z: z_0(s_{h0}s)^m z_1(s_{k0}s)^n z_2$, (where z_0 is either null or s, z_1 is the substring of X' required to make Z a substring of X', and z_2 is either null or s_{k0}) that carries control from q_π to q_π. Note that s_{k1} is not the final symbol of Z.

Let y_0y_1 be any string accepted by G that causes control to be at q_π after deleting y_0. Then the string y_0ZZy_1 is also accepted by G, since after deleting ZZ control is back at q_π. But this string is not accepted by F_r, since ZZ requires control to travel from the $(s_{k0}s)$-loop to the $(s_{h0}s)$-loop without encountering s_{k1}.

Thus, we have exhibited a string not accepted by F_r that must be accepted by G if every string accepted by F_r is also accepted by G. Thus, G cannot be I/O equivalent to F_r. But then $F_r \not\leq_E G$. Thus, F_r is not elementally reducible to any $(1,1)PC_r$-chart.

This completes the proof of the theorems referred to in the text,