Applications of High Level Control Flow

Barry K. Rosen

Computer Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT. Control flow relations in a high level language program can be represented by a hierarchy of small graphs that combines nesting relations among statements in an ALGOL-like syntax with *relevant* perturbations caused by **goto** or **leave** statements. Applications of the new style of representation include denotational semantics, data flow analysis, source level compiler diagnostics, and program proving.

1. INTRODUCTION

Of course we can describe control flow in a program by tracing a path in the control flow graph of the program. But what is "the" graph of a program in an ALGOL-like high level language? How is it related to the source text? Can **goto** or **leave** statements be handled smoothly? How can *sets* of paths that are "similar" from the user's viewpoint be concisely described? Suppose the program is changed. How can we update the graph and the results of analysis by a program prover or an optimizing compiler? Must we start from scratch? Questions like these have led to a new representation of control flow in high level language programs.

Section 2 reviews the standard low level flowchart representation of control flow and contrasts it with the use of a *high level control flow graph* and *induced graphs*. Some applications of the new style of representation are sketched in Sections 3–7. Section 3 shows how denotational semantics can cope with **goto** while avoiding continuations. Section 4 globalizes local data flow information and computes it bottom-up. Section 5 applies the new local information to global problems. Section 6 applies the previous sections to the problem of generating concise but informative compiler diagnostics at source level [FO76; OF76]. Section 7 extends Owicki's method for proving partial correctness of asynchronous parallel programs [Ow75; Ow76; OG76] to cope with **goto** and **leave**. In the special case of sequential programs, a similar but less general trick is proposed by Wang [Wa76]. High level control flow transcends the restriction to classical structured programming that formerly limited the usefulness of axiomatic systems for program proving in the style introduced by Hoare [Ho69]. (The restriction is only partially relaxed in [CH72].)

Two principles that have guided this work should be mentioned. The first is that COMPUTERS ARE NOT PEOPLE: there is little reason to expect computer processing of large structures to be expedited by representations that are natural when people process small structures by hand. For example, compare infix with postfix notation. In this example computational simplicity correlates poorly with intuitive naturalness, but it correlates well with mathematical elegance in the rules defining the formal notation. We have chosen to relate program texts to graphs in a way that is mathematically elegant (though perhaps unnatural for hand processing of small programs) in the hope of expediting computer processing of large programs. The example program SUMFAC in the next section is small enough for hand processing and is just barely large enough to illustrate the points we wish to make. (Another program in this size range is cited in Section 7.) As is shown in Section 6, high level control flow is convenient for communication between a computer and a person as well as for processing by a computer alone.

The second principle guiding this work is a cautiously optimistic version of Murphy's Law: WHATEVER CAN GO WRONG WILL, BUT NOT OFTEN. In particular, well written programs in well designed languages will have few if any **goto** statements, but even good programmers will use them now and then [Kn74]. We conclude that methods for proving or optimizing programs should be able to handle any use of **goto**, but that this ability should not contribute to the processing costs for the many programs free of **goto**. Moreover, the complexity of our methods should rise gradually when anomalous statements are added to a large program. There should not be a drastic change when a single **goto** or **leave** is added to a large program that previously used only single entry and exit control facilities. Such *robustness* is acheived by a mathematical approach that superficially resembles the informal control flow diagrams of [DDH72, Sec. I.7] for classically structured programs but also copes with nonclassical control flow. Ledgard and Marcotty [LM75, Sec. 6] challenge our assumption that classical structured programming is sometimes inadequate. Space does not permit a full discussion of the issues here. See [DEL76] for some recent results that support our assumption.

2. ASSIGNING GRAPHS TO PROGRAMS

The usual low level flowchart representation [Ul73, Sec. 1] is constructed by translating a program into an intermediate text. Unlike the *statements* of an ALGOL-like language with structured programming facilities, the *instructions* in intermediate text cannot be nested within each other. Certain sequences of instructions called *basic blocks* are grouped together to form the nodes of the graph. This representation is a hierarchy with a graph at the top level and sequences of instructions at the level below. An example is shown in Figure 1 for a program SUMFAC written in ALGOL-like notation (supplemented by line numbers) atop the next column.

Unlike the usual shallow hierarchy, the high level representation proposed here has the same depth as the depth of nesting of statements within statements in an ALGOL-like syntax. All places in the hierarchy are held by graphs, and each graph represents a portion of the control flow information in the program. These

```
01 SUMFAC: begin dcl MS, S integer external;
02    comment S will be assigned the sum of the factors of MS;
03    dcl M, P, T, Q, R integer;
04    M := MS; S := 1; P := 1;
05    NEWPRIME: P := P + 1; T := P;
06    AGAIN: Q := quo(M, P); R := rem(M, P);
07    TEST: if R = 0
08       then ZERO: [ T := mul(T, P); M := Q; MORE: goto AGAIN]
09       else NONZERO:
10          [CON1: if T ≠ P then S := mul(S, quo(T−1, P−1));
11           SEMICON: if P>Q then
12             PGQ: [CON2: if M > 1 then S := mul(S, M+1);
13                   S := S − MS;
14                   DONE: leave SUMFAC];
15           ANOTHER: goto NEWPRIME]
16 end
```

graphs are related to each other by means of the *high level control flow graph* for the entire program. This graph is used by the theory to relate high level analysis to concepts usually defined only for low level flowcharts. The smaller graphs in the hierarchy are called *induced graphs.* Semiformal descriptions of the high level control flow graph for a program are in [Ro76a, Sec. 2; Ro77, Sec. 2]. Induced graphs are constructed more formally in [Ro77, Sec. 4]. Here the presentation is rigorous and significantly more general. The sets (2.2.1) later in this section are required to be singletons in [Ro77].

Let G be a finite directed graph with a set $\mathbf{N}_G$ of nodes and a set $\mathbf{A}_G$ of arcs. An arc c has a *source* sc and a *target* tc, both of which are nodes. We identify c with the pair (sc, tc). A *nesting structure* for G is a finite partially ordered set Σ with a maximum π, together with a set $N\alpha$ of nodes and a set $A\alpha$ of arcs for each α in Σ. The following properties are required:

$$N\pi = \mathbf{N}_G \text{ and } A\pi = \mathbf{A}_G; \tag{2.1.1}$$

$$\beta \leq \alpha \text{ in } \Sigma \text{ implies } (N\beta \subseteq N\alpha \text{ and } A\beta \subseteq A\alpha); \tag{2.1.2}$$

$$(c \text{ in } \mathbf{A}_G \text{ has sc, tc in } N\alpha) \text{ implies } (c \text{ is in } A\alpha). \tag{2.1.3}$$

For SUMFAC we take Σ to be the set of all <statement> nodes in the parse tree, with $\beta \leq \alpha$ iff tree node β is a descendant of tree

node α. We deliberately confuse statement labels with the tree nodes they identify, so that SUMFAC, TEST, ... are said to be "in" Σ, with SUMFAC = π and ZERO $\leq$ TEST $\leq$ SUMFAC. For each α in Σ for SUMFAC, let

$$N\alpha = \{ \textit{entering } \alpha, \textit{leaving } \alpha \} \cup \bigcup_{\beta < \alpha} N\beta$$

and construct appropriate arcs. For example, there are arcs from *entering* TEST to *entering* ZERO and to *entering* NONZERO, arcs from *leaving* ZERO and from *leaving* NONZERO to *leaving* TEST, and an arc from *entering* MORE to *entering* AGAIN. The example of (2.1) just sketched for SUMFAC is the high level control flow graph for SUMFAC, which combines syntactic nesting with the details of control flow.

A nesting structure *with entrances and exits* consists of G and Σ as in (2.1) together with, for each α in Σ, sets of *designated entrances* and *designated exits*

$$DENTR\alpha \subseteq N\alpha \text{ and } DEXIT\alpha \subseteq N\alpha \qquad (2.2.1)$$

such that, whenever $\beta \leq \alpha$ in Σ,

$$DENTR\alpha \cap N\beta \subseteq DENTR\beta \text{ and } DEXIT\alpha \cap N\beta \subseteq DEXIT\beta. \qquad (2.2.2)$$

Entrance and exit sets are defined by

$$ENTR\alpha = DENTR\alpha \cup \{ tc \in N\alpha \mid c \in A_G \,\&\, \neg\, sc \in N\alpha \}; \qquad (2.2.3)$$

$$EXIT\alpha = DEXIT\alpha \cup \{ sc \in N\alpha \mid c \in A_G \,\&\, \neg\, tc \in N\alpha \}. \qquad (2.2.4)$$

For all statements in SUMFAC we take

$$DENTR\alpha = \{ \textit{entering } \alpha \} \text{ and } DEXIT\alpha = \{ \textit{leaving } \alpha \}.$$

Other entrances and exits are added by escape or jump statements such as the **leave** (used as in [Wu75, p. 145]) and the **goto**'s in SUMFAC. We can find ENTRα and EXITα without constructing G: all we need is the parse tree and the ability to correlate the label in **goto** NEWPRIME with NEWPRIME as a member of Σ.

Given a nesting structure with entrances and exits, consider any α in Σ, n in ENTRα, and p in EXITα. Let $\Pi(\alpha, n, p) = 1$ if

there is a path from n to p in G touching only nodes of $N\alpha$ (2.3)

and $\Pi(\alpha, n, p) = 0$ otherwise. These *path bits* can be computed bottom-up, beginning with choices of α that are minimal in Σ. Path bits for α can be determined from previously computed path bits for *parts* β of α, where

$$PART\alpha = \{ \beta < \alpha \mid \text{No } \gamma \text{ in } \Sigma \text{ has } \beta < \gamma < \alpha \}. \qquad (2.4)$$

For example, PARTα for α = TEST is { ZERO, NONZERO }.

As in [Ro77, Sec. 4], we construct the induced graph Gα for each α in Σ. The set of nodes in Gα is

$$NG\alpha = N_0\alpha \cup \bigcup_{\beta \in PART\alpha} (ENTR\beta \cup EXIT\beta) \qquad (2.5.1)$$

where

$$N_0\alpha = N\alpha - \bigcup_{\beta \in PART\alpha} N\beta. \qquad (2.5.2)$$

For SUMFAC, $N_0\alpha$ = { *entering* α, *leaving* α }. The set of *real* arcs of Gα is

$$RAG\alpha = \{ c \in A\alpha \mid sc, tc \in NG\alpha \}. \qquad (2.5.3)$$

For each β in PARTα there is a set of *imaginary* arcs

$$IAG[\beta] = \{ (n, p) \mid n \in ENTR\beta \,\&\, p \in EXIT\beta \,\&\, \Pi(\beta, n, p) = 1 \}. \qquad (2.5.4)$$

The total set of arcs is

$$AG\alpha = RAG\alpha \cup \bigcup_{\beta \in PART\alpha} IAG\alpha[\beta] \qquad (2.5.5)$$

with the obvious definitions of sources and targets. (Some arcs are both real and imaginary.)

Let α_1 be the statement CON1 with one part β_1. Then Gα_1 is shown in Figure 2 for i = 1. Now let α_2 be CON2, so that Gα_2 is shown in Figure 2 for i = 2. More generally, if α is any one part conditional statement **if**...**then** β in any program, and if β has only *entering* β as an entrance and *leaving* β as an exit, then Gα is as shown in Figure 2. Similarly for the other control structures of classical structured programming [DDH72, Sec. I.7]. *Escapes* like **leave** and *jumps* like **goto** may be used within β or elsewhere in the program: Gα only depends on the relation between α and its parts. The induced graph construction automatically determines whether Gα is like the simple diagrams of [DDH72, Sec. I.7] or whether the

escapes and jumps in the program are relevant to α. For example, $G\alpha$ for α = SEMICON is as shown in Figure 3.

Figure 4 shows $G\pi$ for π = SUMFAC, and it is roughly twice as large as Figure 1. But Figure 1 presupposes the control flow analysis among instructions that identifies basic blocks and deletes anything not reachable from the entrance to SUMFAC. Similar analysis of Figure 4 leads to the *compressed* induced graph shown in Figure 5. In general, whenever an induced graph is unpleasantly large, we can go beyond [Ro77] by compressing it according to rules like those used to construct low level flowcharts. The formal construction follows.

Given a nesting structure with entrances and exits for a graph G, consider any α in Σ and compress the induced graph $G\alpha$ by applying rules until no further changes are possible. If n is a node in $G\alpha - EXIT\alpha$ not reachable from at least one entrance to α, then

$$\text{delete n and all arcs c with sc} = n \text{ or tc} = n. \qquad (2.6)$$

A node n in $G\alpha$ is *flow trivial* iff both

$$\neg\, n \in ENTR\alpha \cup EXIT\alpha \qquad (2.7.1)$$

and there are unique arcs x, y with

$$tx = n \text{ and } sy = n. \qquad (2.7.2)$$

If n is flow trivial then we bypass it:

$$\text{delete n, x, y and add an arc z from sx to ty.} \qquad (2.7.3)$$

Iterating the bypass operation (2.7) is like grouping instructions into basic blocks.

3. DENOTATIONAL SEMANTICS

It is natural to think of a program statement α as "meaning" a (partial) function from storage states to storage states. In classical structured programming the meaning of α is quite elegantly determined by the meanings of its parts and by the production applied to generate α. The classical case is important but not universal: practical structured programming [Kn74; Wu75; Za74] requires escapes and perhaps a few jumps. In a large program with a few jumps it should be possible to assign meaning bottom-up in an "almost" classical way, without a major change in semantic style such as introduction of continuations [SW74]. Such a *robust* semantic style is easily acheived with the help of Section 2.

The key is simple: instead of asking what α means, we ask what (α, n, p) means, where n is an entrance to α and p is an exit from α. In practical structured programming the number of such triples is often larger than $|\Sigma|$, but not drastically larger. For many of these triples the meanings follow from elementary observations. In general, $\Pi(\alpha, n, p) = 0$ implies that (α, n, p) computes the empty map. In many programming languages the syntax is such that

$$(p \in EXIT\alpha \text{ and } sc = p) \text{ implies } \neg\, tc \in N\alpha, \qquad (3.1)$$

so that (α, p, p) computes the identity map if p is in $ENTR\alpha \cap EXIT\alpha$. In particular, only when $\Pi(\alpha, n, p) = 1$ and $n \neq p$ is there any question about the meaning of (α, n, p) in SUMFAC. The 23 statements give rise to 24 triples in need of analysis here.

Now consider any graph G together with a storage state transformation for each arc and a nesting structure with entrances and exits that satisfies (3.1). We assume determinism: arcs with a common source are assigned transformations with disjoint domains. Given the induced graph $G\alpha$, the transformation for each real arc in $G\alpha$, and previously determined meanings for all triples (β, n_o, p_o) such that β is a part of α, we find that the meaning of (α, n, p) as a storage state transformation is determined. The usual least fixpoint considerations are required if $G\alpha$ has cycles. Transformations are partial functions and, when ordered by inclusion between functions as sets of ordered pairs, have appropriate completeness properties. See [Ma74, Ch. 5; MR76] for details on the mathematical machinery. In particular, the partial functions form a coherent [MR76, Def. 5.7] poset.

4. LOCAL DATA FLOW ANALYSIS

For analysis to support a correctness proof or an optimization it is important to compute meanings that are much simpler than

storage state transformations. For example, we could let the meaning of (α, n, p) be the set of variables whose values can be modified by an execution of α that enters at n and leaves at p, where "can" refers to all control flow paths. Call this set MOD(α, n, p). The previous section applies equally well to this kind of meaning.Moreover, consider CON1 and CON2 in SUMFAC with induced graphs shown in Figure 2. It is easy to show that

$$\text{MOD}(\alpha_i, \textit{entering } \alpha_i, \textit{leaving } \alpha_i) = \text{MOD}(\beta_i, \textit{entering } \beta_i, \textit{leaving } \beta_i). \quad (4.1)$$

The *work* of deriving (4.1) need not be repeated for i = 1, 2 in SUMFAC and for each program. At proof time or compile time we need only *recognize* that α: **if**...**then** β is a one part conditional statement with α related to β in the manner of classical structured programming. Then the meaning of α can be derived from the meaning of β by plugging into the equation

$$\text{MOD}(\alpha, \textit{entering } \alpha, \textit{leaving } \alpha) = \text{MOD}(\beta, \textit{entering } \beta, \textit{leaving } \beta) \quad (4.2)$$

derived at the time of language definition, when we draw Figure 2 without the i subscript. Now consider SEMICON in SUMFAC with induced graph shown in Figure 3. Escapes and jumps perturb the induced graph, but only mildly. The statement SEMICON is *semiclassical* [Ro77, Sec. 4], and the sets MOD(SEMICON, *entering* SEMICON, p) for p = *leaving* SEMICON and for p = *entering* DONE can still be found by plugging into equations derived at the time of language definition.

In general, questions as to what statements can or cannot do to variables or expressions are *local* data flow questions. They can be asked of all statements, not just the minimal ones in Σ that correspond roughly to instructions in an intermediate text. Local data flow questions ultimately refer to the high level control flow graph G and to strictly local information about what can or cannot happen when control flows along each arc. We can answer many such questions by considering the smaller induced graphs and computing bottom-up. When a semiclassical statement is encountered we can plug into a known equation without even considering the induced graph. We used the elementary example of MOD and one part conditionals to illustrate the method. The potential savings are more apparent when the equations like (4.2) for various questions and control operators are considered. Details are in [Ro77] for three important questions: MOD (as above), PRE (the variables that can be preserved), and USE (the variables that can be used).

Of the 23 statements in SUMFAC, only π = SUMFAC fails to be semiclassical. Only for π do we need to trace paths in an induced graph at proof time or compile time. For the same reasons that the usual low level analysis techniques can use basic blocks rather than instructions as nodes, we can analyze the compressed induced graph instead. Whatever graph analysis technique one would use in Figure 1 to determine whether SUMFAC can modify the value of its input MS, the same technique can be used in Figure 5 instead, which is only half as large. With due caution, the local information derived for SUMFAC can then be applied to find local information about calls on SUMFAC in other programs. Recursive procedures lead to a chicken/egg problem: there are circular dependencies in the family of equations like (4.2) derived for a program with recursive procedures. It can be shown that correct and sharpest possible MOD, PRE, USE information is obtained by finding the least fixpoint of the family of equations. Details and a full discussion of the pitfalls are in [Ro76b].

5. GLOBAL DATA FLOW ANALYSIS

Local data flow information was globalized in Section 4. When local information is available for all triples (α, n, p) then high level versions of traditional global flow problems can be solved rapidly. As with low level flowcharts, a global problem is an attempt to assign information to nodes that summarizes either what can happen on some paths or what must happen on all paths. The set of paths considered, for each node n in the high level control flow graph, is either the set of paths to n from nodes in DENTRπ or the set of paths from n to nodes in DEXITπ. (Recall (2.2.1).) The details for live variables are in [Ro77, Sec. 11], and it is clear that traditional constant propagation or available expressions [HU75, Sec. 5.1] can be similarly handled. The more ambitious semilattice versions of these problems [GW76; KU76] are under study. As might be expected, situations where $f(x \wedge y) \neq f(x) \wedge$

f(y) are delicate. Such *nondistributivity* is fairly common [KU76, Sec. 6], and it changes = to ≥ in the Induced Graph Theorem of [Ro77, Sec. 4]. Despite this technical hurdle, we can find an *acceptable assignment* [GW76, p. 177] of information to nodes in the high level control flow graph by calculations with induced graphs only. We are now attempting to generalize the symbolic analysis typified by (4.2) so as to move much of the work backward from compile time to the time of language definition, as is done in [Ro77] for less ambitious flow problems.

A new compiler style is beginning to emerge [Ca77; Ha77; Kn74; Lo76]. Source text and machine code will both appear as dialects of an intermediate language incorporating high and low level constructs. Compilation will consist of gradual expansion from relatively high to relatively low level within the intermediate language, with frequent interludes of analysis and optimization. In particular, special case code generation will be unnecessary because systematic optimization at all levels will do at least as well [Ca77]. These ambitious compilers will update data flow information to reflect program changes due to optimization, so as to polish the code nearly as well as a good programmer could. The syntax directed high level analysis sketched here is particularly oriented toward such compilers. We try to hold down the *total* cost of initial analysis and later updating for well written programs. The well known cost bounds deal only with initial analysis, and no explicit bounds on total cost are available for any data flow analysis method. Another hindrance to a direct cost comparison becomes apparent in the next section. High level analysis yields a great deal of information that is useful for diagnostics. To extract this information from the results of low level analysis requires more work.

6. COMPILER DIAGNOSTICS

Much of the information accumulated so far is of obvious usefulness in understanding and maintaining a program. The ENTR and EXIT sets in (2.2) and the path bits π in (2.3) display the effects of any departures from classical structured programming. They do this concisely at source level for each statement. A low level flowchart like Figure 1 conveys the control flow between basic blocks in one lump that has no simple correlation with the source text. For example, the statement TEST includes seven nodes in Figure 1 and *part* of an eigth. Even though TEST is rather near the root of a large parse tree, it cannot be correlated with the usual shallow hierarchy unless we open up a node to reveal the sequence of instructions within a block:

$$[\ Q \leftarrow \mathbf{quo}(M, P)\ \#\ R \leftarrow \mathbf{rem}(M, P)\ \#\ R = 0?\ ]$$

In practice the situation will often be worse because programs will be larger and will have flowcharts that cannot be neatly arranged on a page.

The local and global data flow information from Sections 4 and 5 is clearly useful for diagnostic purposes or for program proofs [Ro76a, Sec. 6]. Fosdick and Osterweil [FO76; OF76] explain how to reinterpret global flow problems such as live variables to detect data flow anomalies such as an uninitialized use of a local variable. As they imply, it is helpful to warn the programmer that B := A+2 can be reached before A is initialized, but much more helpful to couple the warning with an example path whereon this happens.

Now suppose that B := A+2 occurs in the context

CONTEXT: [FIDDLE: **if**...**then** A := 0 **else** MESS: [...];
THICKET: [...];
CHANGE: B := A+2]

where MESS and THICKET are large and intricate statements such that, for γ = MESS and for γ = THICKET,

$$\neg\ A \in \mathrm{MOD}(\gamma, \textit{entering}\ \gamma, \textit{leaving}\ \gamma).$$

To trace an anomalous path from a point corresponding to *entering* CONTEXT to a point corresponding to *entering* CHANGE in a low level flowchart would be confusing. Irrelevant details of paths through MESS and THICKET would be included. High level languages should relieve the programmer of the need to read low level programs, at least when fine tuning of run time performance is not an immediate issue. The exclusive use of low level control and data flow analysis methods precludes such relief, unless we choose to have data flow diagnostics that are much less informative than is desirable. Programmers can cope with exclusively low level diagnostics, but "there is little appeal in a miraculous new language, if

the corresponding debugging tool talks back to me in octal or hexadecimal" [Sp76, p. 294]. (Similar remarks apply to program proving when escapes and jumps are permitted.)

In the induced graphs $G\alpha$ for α = CONTEXT and α = FIDDLE we can easily and naturally display short paths that characterize all the anomalous paths in the full high level control flow graph. We begin in CONTEXT with the path

from *entering* CONTEXT to *entering* FIDDLE
to *leaving* FIDDLE (along an imaginary arc)
to *entering* THICKET
to *leaving* THICKET (along an imaginary arc)
to *entering* CHANGE,

where the second imaginary arc cannot modify A. The first imaginary arc can modify A, but it can also preserve A. Therefore we provide more detail in FIDDLE with the path

from *entering* FIDDLE to *entering* MESS
to *leaving* MESS (along an imaginary arc)
to *leaving* FIDDLE,

where all arcs cannot modify A. High level analysis leads directly to concise but informative diagnostics at source level when the methods of [FO76; OF76] are applied to languages amenable to practical structured programming.

7. PROGRAM PROVING

It is convenient to consider the original values of input variables as persisting in a special part of the storage state ξ of a computation. Then *partial correctness* [Ma74, Sec. 3-1] with respect to predicates P, Q on storage states is the property that, whenever a computation begins with ξ such that $P(\xi)$ and finishes with η , then $Q(\eta)$. Owicki [Ow75; Ow76; OG76] proposes a powerful method for proving partial correctness of asynchronous parallel programs. Two assertions, a precondition and a postcondition, are associated with each α in Σ, as in the style of proof introduced by Hoare [Ho69]. Carefully chosen assertions can lead to clear and convincing proofs despite the pitfalls of asynchronous parallelism. We reformulate a crucial point more abstractly and much more generally, so as to extend the applicability of the method.

We assume that the maximum π in Σ satisfies

$$\mathrm{DENTR}\pi = \{\ \textit{entering}\ \pi\ \} \text{ and } \mathrm{DEXIT}\pi = \{\ \textit{leaving}\ \pi\ \}, \qquad (7.1)$$

but none of the other constraints of classical structured programming are needed. A storage state predicate C[n] is to be assigned to each node n in the high level control flow graph G, in such a way that, for all states ξ, η and arcs c in G:

$$P(\xi) \text{ implies } C[\textit{entering}\ \pi](\xi); \qquad (7.2.1)$$

$$C[\textit{leaving}\ \pi](\eta) \text{ implies } Q(\eta); \qquad (7.2.2)$$

$$(C[sc](\xi) \text{ and } \xi \rightarrow_c \eta) \text{ implies } C[tc](\eta), \qquad (7.2.3)$$

where $\xi \rightarrow_c \eta$ means that ξ can be changed to η when control flows along c. In sequential programming this would guarantee partial correctness, and we have so far just transcribed [Fl67] into the notation of Section 2. Now we impose the *interference-free* condition [Ow75, Sec. 3.3]: for all states ξ, η and all arcs c in G such that c and n are in distinct parallel processes,

$$(C[n](\xi) \text{ and } C[sc](\xi) \text{ and } \xi \rightarrow_c \eta) \text{ implies } C[n](\eta). \qquad (7.3)$$

An assignment C from nodes to assertions that satisfies (7.3) as well as the Floyd conditions (7.2) will assure partial correctness. In [Ow76; OG76] the sharing of variables by processes is restricted in ways that expedite verification of (7.3).

For us the precondition and postcondition for α in a Hoare style proof are $C[\textit{entering}\ \alpha]$ and $C[\textit{leaving}\ \alpha]$. By directly assigning one assertion to each node in G rather than assigning two assertions to each statement in Σ, we transcend the restriction to classical structured programming. Wang [Wa76] proposes a similar trick for sequential programs in the case where each α in Σ, not just π, satisfies (7.1). Only if $N_0\alpha$ in (2.5.2) is always $\{\ \textit{entering}\ \alpha, \textit{leaving}\ \alpha\ \}$ will [Wa76] assign assertions to all nodes of G.

Consider the program FINDPOS [Ro76a, Sec. 3] that has been used to illustrate what is so hard about asynchronous parallelism. Owicki presents an elegant partial correctness proof for a similar program Findpos [Ow75, Fig. 3.4] wherein the effects of **leave** statements are simulated in classical structured programming. The

program Findpos is a shade less clear and distinctly less efficient than FINDPOS. The mathematical machinery in [Ow75] cannot say, much less prove, that partial correctness of Findpos implies partial correctness of FINDPOS. With (7.2) and (7.3) however, we can prove the partial correctness of FINDPOS directly by an easy adaptation of [Ow75, Fig. 3.4]. The formulation (7.3) of interference-freeness is also simpler than the original one. Adding [Ro76a, Lemma 5.1] to the partial correctness proof leads to a total correctness proof more elegant than [Ro76a, Sec. 5].

REFERENCES

Ca77. Carter, J.L. A case study of a new code generating technique for compilers. *Comm. ACM* (1977?) to appear.

CH72. Clint, M., and Hoare, C.A.R. Program proving: jumps and functions. *Acta Informatica* **1** (1972), 214-224.

DDH72. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming.* Academic Press, London and New York, 1972.

DEL76. DeMillo, R.A., Eisenstat, S.C., and Lipton, R.J. Can structured programs be efficient? *SIGPLAN Notices* vol. 11 num. 10 (October 1976), 10-18.

Fl67. Floyd, R.W. Assigning meanings to programs. *Proc. Symp. Appl. Math.* **19** (1967), 19-32.

FO76. Fosdick, L.D., and Osterweil, L.J. Data flow analysis in software reliability. TR CU-CS-087-76, Computer Science Dept., University of Colorado, Boulder, May 1976.

GW76. Graham, S.L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. *J. ACM* **23** (1976), 172-202.

Ha77. Harrison, W. A new strategy for code generation — the general purpose optimizing compiler. *These Proceedings.*

HU75. Hecht, M.S., and Ullman, J.D. A simple algorithm for global flow data flow problems. *SIAM J. Computing* **4** (1975), 519-532.

Ho69. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM* **12** (1969), 576-583.

KU76. Kam, J.B., and Ullman, J.D. Global data flow analysis and iterative algorithms. *J. ACM* **23** (1976), 158-171.

Kn74. Knuth, D.E. Structured programming with **goto** statements. *Computing Surveys* **6** (1974), 261-302.

LM75. Ledgard, H.F., and Marcotty, M. A genealogy of control structures. *Comm. ACM* **18** (1975), 629-639.

Lo76. Loveman, D.B. Program improvement by source to source transformation. *Proc. 3rd ACM Symp. on Principles of Programming Languages* (January 1976), 140-152.

Ma74. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

MR76. Markowsky, G., and Rosen, B.K. Bases for chain-complete posets. *IBM J. Res. and Devel.* **20** (1976), 138-147.

OF76. Osterweil, L.J., and Fosdick, L.D. DAVE: a validation, error detection, and documentation system for FORTRAN programs. *Software Practice and Experience* (1976) to appear.

Ow75. Owicki, S.S. Axiomatic proof techniques for parallel programs. Tech. Rep. 75-251 (Ph.D. thesis), Computer Sci. Dept., Cornell U., Ithaca New York, July 1975.

Ow76. Owicki, S.S. A consistent and complete deductive system for the verification of parallel programs. *Proc. 8th Ann. ACM Symp. on Theory of Computing* (May 1976), 73-86.

OG76. Owicki, S.S., and Gries, D. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM* **19** (1976), 279-285.

Ro76a. Rosen, B.K. Correctness of parallel programs: the Church-Rosser approach. *Theoretical Computer Science* **2** (1976), 183-207.

Ro76b. Rosen, B.K. Data flow analysis for procedural languages. IBM Research Report RC 5948, Yorktown Heights, April 1976.

Ro77. Rosen, B.K. High level data flow analysis. *Comm. ACM* (1977?) to appear.

Sp76. Spier, M.J. Software malpractice — a distasteful experience. *Software Practice and Experience* **6** (1976), 293-299.

SW74. Strachey, C., and Wadsworth, C.P. Continuations: a mathematical semantics for handling full jumps. Tech. Mono. PRG-11, Programming Res. Grp., Oxford U., January 1974.

Ul73. Ullman, J.D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica* **2** (1973), 191-213.

Wa76. Wang, A. An axiomatic basis for proving total correctness of **goto** programs. *BIT* **16** (1976), 88-102.

Wu75. Wulf, W.A., et al. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

Za74. Zahn, C.T. A control statement for natural top-down structured programming. *Lecture Notes in Computer Sci.* **19** (1974), 170-180.

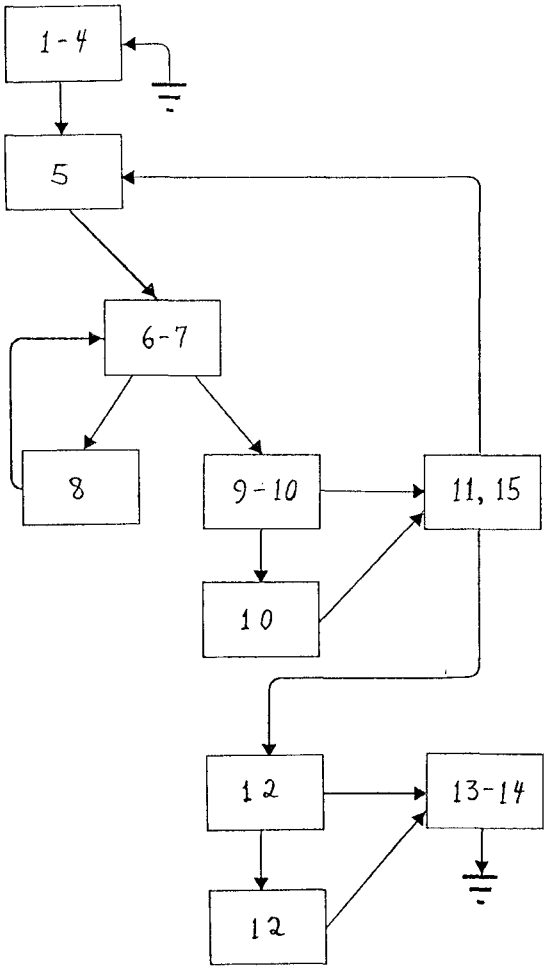

Fig. 1. Low level flowchart for SUMFAC. Each node is a basic block of instructions in an intermediate text (not shown) constructed in the usual way. For each block we indicate which line numbers in SUMFAC give rise to instructions in that block. "Grounded arcs" indicate entrances and exits.

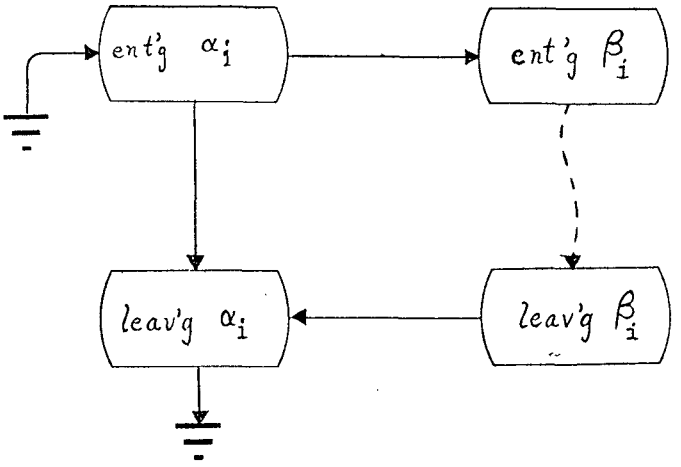

Fig. 2. Induced graph $G\alpha_i$ for α_1 = CON1 and α_2 = CON2 in SUMFAC. The sinuous dashed arc is imaginary. The same picture with no i subscripts would be good for any α: **if** ... **then** β statement in classical structured programming.

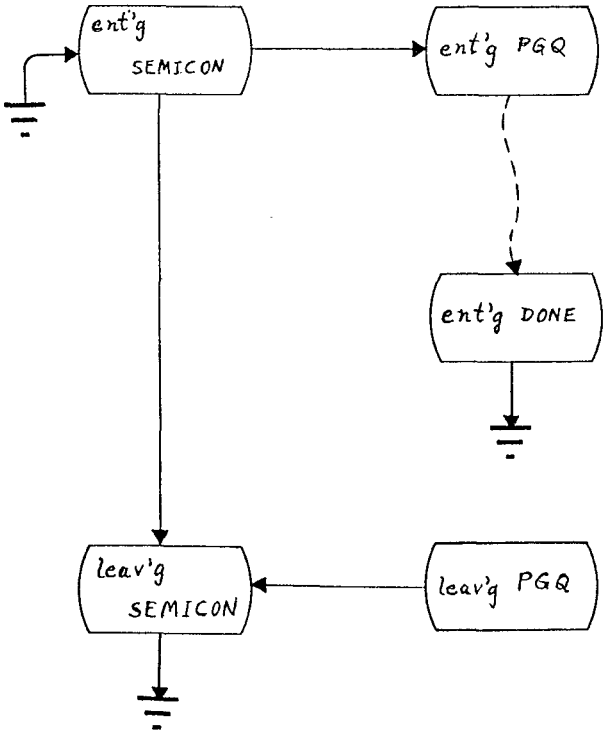

Fig. 3. Induced graph $G\alpha$ for α = SEMICON in SUMFAC. Note that *leaving* PGQ is not reachable from any entrance to SEMICON. This graph and the graphs shown in Figure 2 belong to the infinite family of graphs induced by one part conditional statements α: **if**...**then** β that appear in programs with various uses of escapes and jumps. These graphs also belong to the infinite subfamily derived from one part conditionals that are "semiclassical". In practical structured programming most statements are semiclassical.

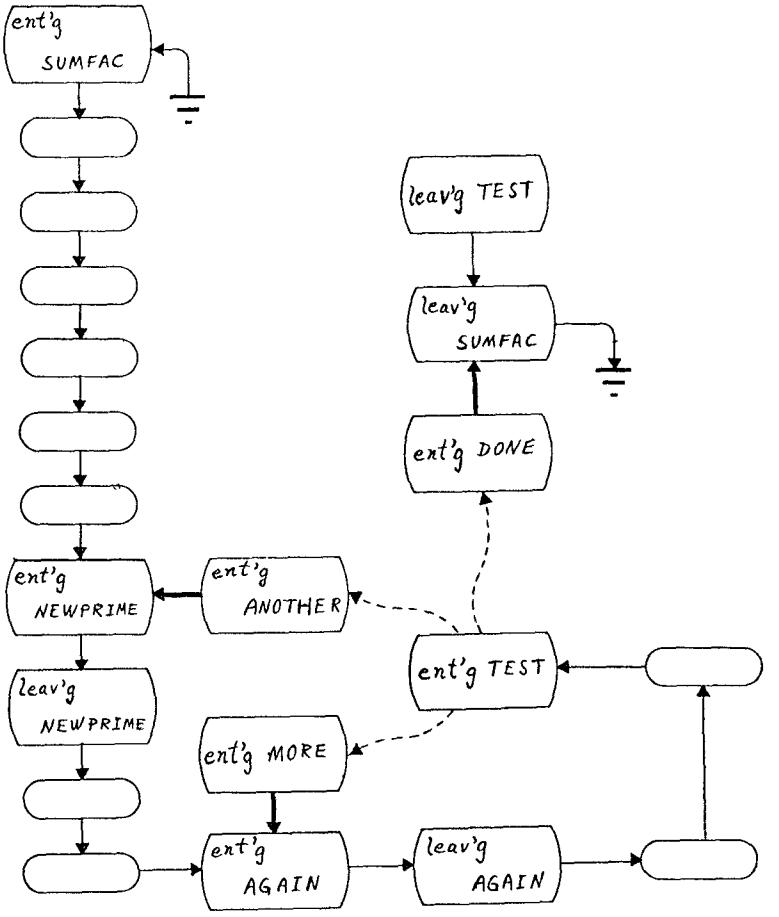

Fig. 4. Induced graph $G\pi$ for π = SUMFAC. Sinuous dashed arcs are imaginary but not real. Heavy arcs are contributed by escapes and jumps.Because two of these arcs pass control from TEST to *other parts* NEWPRIME and AGAIN of π, the statement π fails to be semiclassical.

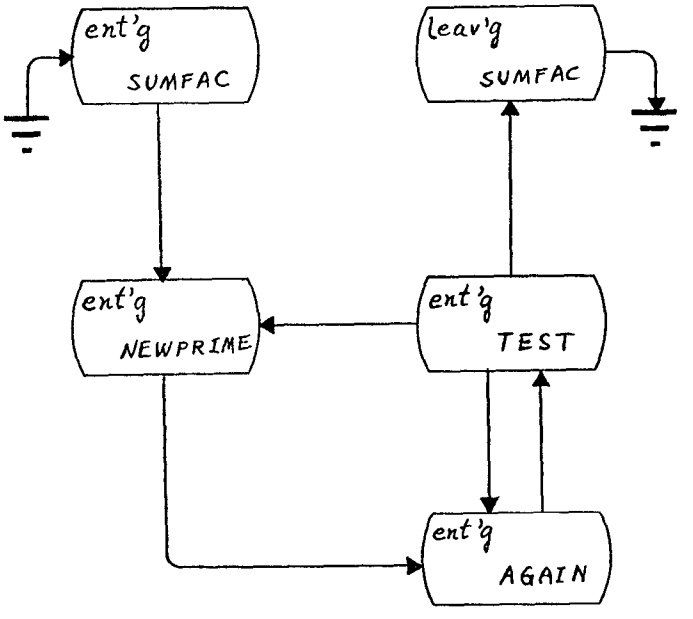

Fig. 5. Compressed induced graph $CG\pi$ for π = SUMFAC. An arc from m to p in $CG\pi$ summarizes the net effect of a path from m to p in $G\pi$ that passes through only "flow trivial" nodes.