The control flow graph (CFG) is most frequently defined after the program has been transformed to some intermediate code representation, usually 3 address code [AHO 85]. Translation into this intermediate code loses structural information (for example N-way branches) that we will be able to use in analysis and in implementing efficient run time support for identifying implicit process groups. We will here extend the definition so we may apply it to more general analysis, including higher level code, specifically code which may be in languages such as Fortran or C. In order to do this we will need to extend the standard definition of basic block to take into account the richer control flow constructs available in higher level languages, and use this extended definition in our CFG.

First we must define what we mean by a block. We can identify blocks of code in a program as a set of statements that are executed consecutively. Examples of this are the statements in the scope of an IF statement or loop. Blocks are frequently designated by textual block delimiters such as matching brackets \{ \} in a C program, or matching THEN, ELSE and ENDF statement in Fortran. Such delimiters are defined as part of the syntax of a language, they are not always required.

**Definition 0.1.** A code block is a set of contiguous statements in a program such that control may only reach a statement in the block by passing through the first statement.

A subroutine or function is always a code block; even though it may be called from multiple points in the program, control normally enters at the first statement (there are exceptions to this in Fortran which we will discuss below). It is possible, however for code blocks to be nested inside other code blocks. For example, the body of a natural loop is a code block. It may contain an IF-THEN-ELSE-ENDIF statement, which delimits two code blocks: one which executes if the predicate of the IF statement evaluates to TRUE, and the other if it is FALSE.

To define a control flow graph (CFG) we will indicate control transfers from one block to another with arcs, but since code blocks may be nested, they cannot serve as nodes. We will define a basic code block, or simply basic block, as the largest group of statements that we could graph as a single node. The following definition extends the standard definition of [AHO 85] by adding communication statements and block delimiters to the definition of leader.

**Definition 0.2.** A basic block is a set of executable statements in a program that must always be executed from start to finish or not at all. Control may only enter at the first statement, called the leader, of the basic block, and if execution reaches the first statement, then it must continue to the last statement of the block.

Leaders are identified as follows (based on Aho & Ullman):

1. The first executable statement of the program is a leader.
ii. Any executable statement which is the target of a (conditional or unconditional) jump or control transfer is a leader.

iv. Any executable statement which immediately follows a conditional statement is a leader.

v. Any executable statement following a block delimiter is a leader.

For purposes of analyzing parallel programs, we will also define:

vi. Any synchronization or communication statement is a leader.

(Intuitively, a data transfer occurs at a synchronization, which changes the data set at each process. We are going to define state of a process in terms of data in memory and basic block being executed; data received from another process implies a change of the serial state at each process that cannot be determined from data at that process.)

The body of a basic block is the set of statements in the text starting with a leader and up to but not including the next leader, or up to the end of the program text if no leader is encountered.

Note that the above definition implies that there can be only one communication statement in a basic block; we start a new block whenever we encounter such a statement. The reasoning behind this is that communication statements can change the data set unpredictably at the communicating process. We take something that changes the state in some way that is not determined by the data present in the process at entry to the basic block as changing also the basic block. The same reasoning applies to synchronization statements, which might involve some data or information transfer. The key point in the basic block is that its execution is totally predictable from the contents of memory on entry to the block and the position of the block in the CFG. We are only concerned with executable statements for the purpose of defining basic blocks, but we include (as executable) statements such as continue in Fortran which specify no operation, but may be targets of conditional or unconditional jumps, or indicate the range of a loop.

Logic statements in Fortran or C include added text to delimit the range of the statement. For example, the IF statement may be followed by else, and endif in Fortran, or by a block of code enclosed in brackets in C. The IF statement itself follows the rules for conditionals and ends the preceding block; the others are delimiters; they should not be considered executables, but they do define blocks of code.

Loops in higher level languages require special consideration, due to the complexity of control structures. In general, the first statement in a loop is a leader, and the last statement in the loop ends the final basic block. Statements such as do[... ] in C should be considered textual delimiters of a block of code rather than executables. However, statements such as while(), until() and for() are logic statements and end a basic block. If such are at the end of a loop (e.g. do[... ]while(); ) they may be considered part of the final block in the loop, since execution of that final block always continues through the statement in question.

Loop control statements at the beginning of a loop are basic blocks unto themselves. Since they may prevent execution of the whole loop, they cannot be part of the first block inside the loop. Since control passes from inside the loop back to the first statement to determine if the next iteration should be executed, this statement cannot be part of the block just before the loop. Loop control statements may be
Figure 0.1. Skeleton Fortran program with basic blocks identified. Direct analysis of high level code creates a different block structure than analysis of intermediate code. For example, block 2 would translate (on most hardware) to a pair of conditional jumps and thus be split into two blocks.
at the same time targets of jumps and conditionals, so they meet the requirements for start and end of a basic block.

The Fortran DO loop statement may apply up to the next END DO, which may be considered a textual marker (like brackets in C). There are alternate methods of specifying the scope of a loop in Fortran, such as ending on a numbered statement. We will here limit ourselves to consideration of loops which could be written with DO ... END DO, and avoid complications that may arise with older versions of Fortran which allow jumps into the body of a loop, or to its last statement.

It is possible in Fortran to have a set of nested loops which each use the same numbered statement as the last statement in the loop. If the statement performs no operation, as in a CONTINUE statement, then a single label can be replaced, for purposes of analysis, with a set of END DO statements. If the statement is an executable operation, we should consider it to belong to the innermost loop, which, being executed by all the outer loops, makes the statement execute inside all loops.

Definition 0.3. A program control flow graph, or CFG, is a directed graph: $F = \{V, A, s, E\}$

in which the vertices $V$ are the basic blocks in the program and the arcs $A$ denote control transfer from the end of one basic block to the beginning of another. $E$ is the set of end nodes, in which the program may halt and the node $s \in V$ is the first basic block in the program, where execution always begins.

We have added a start node and a set of end nodes to the conventional definition of CFG because we will require them in our analysis. We do not believe that this imposes a serious restriction on the set of programs that can be thus analyzed. While it is possible to imagine a program, such as an operating system, which is supposed to loop for an indefinite period, it should not be seriously restrictive to allow the possibility of a halt instruction which would send it to an end node (such an instruction could be added if not already present).

A subroutine or function call may be represented as a vertex in the CFG. Even though a subroutine may have multiple exit points in the code, it always returns control to the statement immediately following the call, so the call looks to the external program like a basic block. The code of the subroutine itself is then represented by a separate CFG.

It may be imagined that a program could be started at several different nodes, perhaps by using several different program launch routines (note that this is not the case of a program which starts at a single node $s$ and immediately jumps to one of a set of other nodes). It is possible that the same program text started in different places will result in drastically different execution. It is clear that the properties of the program could be drastically different; for example some nodes could be reachable from one start point that are not reachable from another. We will therefore require a single start node, and if we somehow manage to launch the same program code from a different node we will consider this to be a different program.

(An exception to subroutines having single entry points is found in Fortran subroutines that use the entry statement. This allows defining different entry points, call parameters and return types within a single subroutine. The implications of multiple entry points in a subroutine are the same as for a program; i.e. the CFG for the subroutine will in general be different, have different connectivity and data}
dependences. Therefore a subroutine with multiple entries has to be treated as a
different subroutine for each entry statement).

If we have a CFG that corresponds to a program with multiple end nodes we
will need to transform this for purposes of our later analysis to a CFG with a single
end point. This can always be done without changing the meaning of the program:

**Theorem 0.4.** Given a program \( P \), with a CFG, \( F = \langle V, A, s, E \rangle \) in which
\(| E | > 1\), we can transform \( P \) to a program \( P' \) with \( F' = \langle V', A', s, E' \rangle \) and
\(| E' | = 1\); that is, a program with a single exit node, and such that \( P' \) and \( P \)
perform the same computation.

**Proof.** Construct \( P' \) as follows: Add to \( P \) a single end node \( e \) such that \( E' = \{e\} \),
such that the only instruction in \( e \) is a stop or end statement. Replace any stop
or end statements in every node in \( E \) with jumps to \( e \). Every node in \( E \) will then
have an exiting arc to \( e \), and execution will follow that arc in \( P' \) whenever it would
have halted in \( P \). \( P' \) executes the same code as \( P \) until program end, and then only
executes one additional jump which does not modify any variable in memory or
produce any output. Therefore computed results are unchanged.

Given the application of the same algorithm to generate the CFG, we can be sure
that only one CFG corresponds to each program:

0.1. **Reducible CFG.** We now need to define what we mean by a reducible CFG,
for which we will need several other definitions. In most of these we are following
Aho & Ullman [AHO 85].

**Definition 0.5. DOM relation.** Given a CFG \( G = \langle V, A, s, E \rangle \), and nodes
d, n \( \in A \), d dominates n (which may be written \( d \ DOM n \)) if all paths \( s \rightarrow^* n \) are
of the form
\[ s \rightarrow^* d \rightarrow^* n \]
That is, all paths from \( s \) to \( n \) pass through \( d \).

Note that the DOM relation depends on the identification of a start node. Con-
sider, for example, a loop containing several nodes. If we select any one node in
the loop as the start node, it will dominate all the others (the start node always
dominates all reachable nodes). It will be convenient in our development to select
a particular node that is not the start node of the CFG as the start node for DOM.
In such a case, our selected start node will dominate all and only the nodes that are
reachable from it. [AHO 85] describes algorithms for efficiently computing DOM.

Some properties of DOM, proved in [AHO 85] are:

**Lemma 0.6.** DOM is a reflexive partial order

Also:

**Lemma 0.7.** Dominators of a node \( n \) are linearly ordered by DOM.

D dominators may be displayed as a DOM tree, in which the start node \( s \) is the
root node, the parent of each node is its immediate dominator, and all and only
the dominators of a node \( n \) are its ancestors in the tree. The DOM relation may also be
used to classify edges in a CFG:

**Definition 0.8.** A back edge of the CFG is an arc \( a \rightarrow b \) such that \( b \) (the head)
dominates \( a \) (the tail).
That is, back edges follow paths that take us closer to the start node \( s \).

**Definition 0.9.** A **forward edge** of a \( CFG < V, A, s, E > \) is an arc \( a \rightarrow b \) such that there is an acyclic path of the form \( s \rightarrow^* a \rightarrow b \).

Such an arc is a member of a set of edges which, taken together, form an acyclic graph in which every node can be reached from the start node \( s \) of the \( CFG \), the definition in [AHO 85]. That is, if all edges were forward edges, the entire program would be an acyclic graph.

We now introduce a specific class of \( CFG \).

**Definition 0.10.** A **reducible flow graph** is a \( CFG < V, A, s, E > \) in which all edges in \( A \) can be partitioned into:

- **Forward edges**, with the property that the set of all forward edges forms a directed acyclic graph (DAG) in which all nodes are reachable from \( s \), and:
- **Back edges**, of the form \( a \rightarrow b \) such that \( b \) DGM \( a \).

Note that the notion of reducibility is closely related to the presence of cycles in a \( CFG \). An acyclic \( CFG \) is automatically reducible, since all its edges are forward edges. A \( CFG \) in which all loops have single entry points is also reducible, since such loops have graphs in which all nodes are dominated by the entry node. Therefore an edge that loops back to the entry node is a back edge - it is dominated by its head. However, a loop with more than one entry point will not have this property. Consider a loop with two entries: since nodes in it will be reachable on paths through either entry node, neither node dominates the loop. Therefore the edges that form the cycle by looping back to the entry nodes are not back edges, and they are not forward edges either because they form a cycle.

0.2. **Loops in reducible \( CFG \).** We will initially limit our consideration to programs that can be represented by a reducible \( CFG \). All edges in such a \( CFG \) are either forward or back edge, and loops are easily and unambiguously identified in such programs.

**Definition 0.11.** The **natural loop** \( L \) of a back edge \( x \rightarrow s \) is the set of those nodes that have paths to \( x \) without passing through \( s \), and \( s \) itself, called the header of the loop. ([AHO 85], among others, give algorithms for identifying natural loops). Note that \( s \) dominates every node in the loop; that is, there are no arcs to any node in \( L \) from any node outside \( L \) that do not pass through \( s \).

From [AHO 85] we have:

**Lemma 0.12.** In a reducible \( CFG \), all cycles are natural loops.

We will be using the idea of an exit node from a loop in much of our continuing development, so we need to establish the following property:

**Theorem 0.13.** In a natural loop, an exit node has outdegree greater than one, and at least one immediate successor of an exit node is outside the loop.

**Proof.** Assume false. Let there be an exit node \( y \), of outdegree greater than one, such that all paths \( y \rightarrow z \) are such that \( z \) is inside the loop. Let the back edge of the loop be \( x \rightarrow s \). Then all immediate successors \( z \) of \( y \) must have a path \( z \rightarrow x \) without passing through \( s \) (by definition 0.11), and \( y \) is not an exit of the loop. Assume instead that \( y \) is an exit node of outdegree one, and has an immediate
0.3. Loops with multiple exits. Let $L$ be the natural loop of the back edge $x \to s$ (def: 0.11). We know from the definition that we can not have any arc to any node in $L$ except $s$ from a node outside $L$, because $s$ dominates $L$. It is, however, possible to have a back edge $z \to s$, where $z \neq x$.

In this case we can define two natural loops with the same header. Let the natural loop of $x \to s$ be $L$, and the natural loop of $z \to s$ be $M$. Both have the same header, $s$. There are nodes $y \in L \cap M$ from which both $z$ and $x$ may be reached, and also possibly nodes that are in $L$ or $M$ but not in both. The
Definition 0.14. A **compound natural loop** is either the natural loop of a single back edge, or the union of the natural loops of a set of back edges which all point to the same header node. We may simply call these compound loops, recalling however that a simple natural loop is included in the definition (see fig 0.3).

Compound loops may be identified by finding all the natural loops in a CFG and then taking the union of loops that share the header node. A property of a compound loop in a reducible CFG is that if execution enters the loop at the header node, it may only leave the loop through one of the exit nodes of the compound loop. Also, by the definition (0.11) of natural loop, it is not possible to enter the compound loop except at the header node. A process executing along any branch
in the compound loop must return to the header node. Therefore it may select a
different branch for each iteration, and exit on any of the possible exits.

Therefore a compound loop defines a subgraph within the CFG with a single
entry point and N exit points, each of which is a loop exit.

0.4. Non-reducible CFG. Studies of random samples of Fortran programs show that
most such programs do have reducible flow graphs [ALLEN 70],
[KNUTH 71]. These studies show that even when programs are written using
GOTO statements, in a language such as Fortran which is not designed for struc-
tured programming, programmers tend to write reasonably well structured code
which leads to reducible CFG. Even so, we need to be able to deal with a non-
reducible flow graph when it occurs.

A non-reducible CFG is characterized by a cycle in which there is at least one
e edge that is neither a back edge nor a forward edge. That is, its tail does not
dominate its head, which would make it a forward edge, nor does its head dominate
its tail which would make it a back edge. We find that typically this occurs when
we have a cycle with more than one entry point. It is not a natural loop because
there is no node identifiable as the head of the loop.

Consider a cycle with two entry points at different nodes. This can be trans-
formed to a reducible graph by replicating all the nodes in the cycle and connecting
one of the entry arcs to one set of nodes, and the other entry arc to the other. That
is, convert one cycle with two entries into two duplicate cycles each with single
entries. Having done this, the resulting pair of graphs is each a natural loop, and
reducible.

Techniques for converting non-reducible CFG to reducible can be found in [AHO 85]
and other compiler references.

References

[AHO 85]  A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and
Tools, Addison-Wesley 1985

Co-operative Multi-thread Parallel Tabu Search with an application to Circuit
Partitioning, in Solving Irregularly Structured Problems in Parallel, vol-

[ALLEN 70]  F. E. Allen, Control Flow Analysis, SIGPLAN, pp. 1-19, 1970

[ALPERT 96]  R. Alpert, C. Dubnicki, E. W. Felten, K. Li, Design and Implementation
of NX Message Passing Using Shrimp Virtual Memory Mapped Commu-
nication, Proc. of the Internat. Conference on Parallel Processing, August
1996.

[APT 97]  K. R. Apt, E. Olderog; Verification of serial and Concurrent Programs,

[BAGHERI 92]  B. Bagheri, T. W. Clark and L. R. Scott; Pfortran: A Parallel Dialect of

[BAGHERI 94]  B. Bagheri, Parallel Programming with Guarded Objects, Pennsylvania
State University Computer Science, August 1994

[BALA99]  P. Bala, T. W. Clark, Experiences with Co-Amy Pfortran and Pfortran
as Tools for Parallelization of a Large-Scale Scientific Application. Pro-
cedings of the Third International Conference on Parallel Processing and
Applied Mathematics, pp. 367-377. R. Wyrzykowski, B. Mochnacki, H.
Piec, J. Szopa, Editors., Technical University of Czestochowa, 1999

[BSP 96]  D. B. Skillcorn, J. M. D. Hill, W. F. McColl, Questions and answers about
BSP, technical report TR-15-96, Oxford University Computing Labora-
tory, August 1996
Figure 0.4. Non-reducible CFG.
A subgraph from CFG of Figure ??, with arc (3, 7) added. Once this is done, arc (7, 4) is no longer a back edge. Nodes 4, 5, 6, 7 are now a cycle with two entry points (4 and 7), and no longer a natural loop.


[DEO 74] Narsingh Deo; Graph theory with Applications to Engineering and Computer Science, Prentice-Hall 1974.
Figure 0.5. Non-reducible graph transformed to reducible by replicating cycle.


R. Johnson, K. Pingali, *Dependence Based Program Analysis*, SIGPLAN ’93 PLDI, June 1993


[PLANG] Planguages web site: http://planguages.cs.uchicago.edu/


