A New Abstraction for the Study of Module Interconnection

Narayan C. Debnath
Computer Science Department
Winona State University
Winona, MN 55987

Abstract

Graph models have been useful abstractions for examining the programmer's perspective of the static structure of imperative programs. A number of graph models referring to control flow and data dependencies have appeared in the literature. In this paper, a more general abstraction for describing imperative programs is developed. The proposed abstract model, called Generalized Program Graph (GPG), is defined to be a digraph where nodes denote variable definitions and predicates, and edges represent either control flow or data dependencies. Aspects of GPG construction are discussed and an algorithm for building a GPG is outlined. As a major application, GPG is directly useful to represent and analyze software module interconnection structure. A formal model of software module interconnection structure is defined. Because the GPG includes both the control flow and data dependency information in one abstractions, it can be used as an implementation model for the development of measurement and analysis tools for empirical research.

1. INTRODUCTION

Graph models have been useful abstractions for examining the programmer's perspective of the static structure and complexity of imperative programs. The familiar control flow graph has been used to study structured programming, testing methodologies, and develop complexity measures [1-5]. The data dependency graph has been proposed as an abstraction useful for studying the interconnection between data objects in a program and for developing measures of data dependency complexity [6]. Such graph models show promise as a basis for developing a range of useful software design and development tools.

The work described in this paper was motivated by the desire to study the interface between the structure of control flow and data dependencies. The complexity of the control flow and the complexity of the data dependencies are necessarily intertwined. An increase in the complexity of the data dependencies will result from an increase in control flow complexity. A control flow alternation construct can result in an increase in the number of variable definitions that may be referenced. This increase in the number of referable definitions may extend far beyond the range of the alternation construct and this increase cannot be viewed by examining the control flow graph alone [7].

In order to study the interface between control structure and data dependencies, it becomes imperative that a more general abstraction of computer programs must be developed. This paper is primarily devoted to the development of such an abstract model for representing imperative language programs. The abstraction proposed in this paper is called Generalized Program Graph (GPG), and is essentially a synthesis of the control flow graph [1] and the data dependency graph [6]. The GPG representation is structurally equivalent to the string form of a program. However, because of its graph structure, a GPG is better suited for structural analysis of a program than the original string form.

The GPG is defined to be a directed graph with nodes representing variable definitions and predicates, and edges denoting control flow and data dependencies. In order to construct a GPG corresponding to a given program, one must determine live definitions of variables at various program statements. A variable definition is considered "live" at a specified statement if it is possible for the value assigned to the variable at the definition to be referenced at the statement. While drawing data dependency edges in a GPG, live definitions are collected from alternate pathways in the program to determine possible dependencies of a given variable definition or a predicate at a particular statement. Discussions concerning various sources of data dependencies in a program and live definitions of variables are explicitly presented. An algorithm for constructing a GPG from a given source program together with some examples of GPG construction are also included. It appears that the GPG can directly be applied in the studies of software module interconnection complexity. A preliminary model describing the structure of software module interconnection is proposed, and other important practical and theoretical applications of the GPG are discussed.

2. A UNIFIED GRAPH MODEL OF IMPERATIVE PROGRAMS

A program written in an imperative language typically consists of a set of variable definitions, together with the unique order of execution of the definitions governed by control flow constructs. A variable definition is a statement that may modify the value of a variable, such as assignment statements, procedure calls, and input statements. The control flow is the direct representation of the prescribed order of execution of the definitions in the program. The control flow, in its simplest form, prescribes the sequential execution of contiguous statements. However, if there is more than one possible alternative next statement, a predicate is used to specify the choice. A predicate consists of an expression that evaluates to either true or false (assuming only binary predicates), and this boolean result precisely determines the next statement to be executed.

In this section, a rigorous definition of the generalized program graph, an abstraction capturing features of both the control flow graph and the data dependency graph, is provided. Several important aspects of GPG construction is also explicitly discussed.

2.1. Generalized Program Graph

The generalized program graph is an abstract representation of a computer program written in an imperative language. The GPG is a directed graph with each node denoting either a variable definition or a predicate, and edges representing control flow and data dependencies.

Formally, the GPG representation of a program P is defined to be a digraph,

$$GPG = (N, E, s, f),$$

where N is a finite set of nodes, E is a set of edges in N x N, and s, f ε N. Nodes s and f represent the unique entry node and exit node respectively. In addition,

$$N = N_P \cup N_D \cup s \cup f,$$

where
(a) there is a one-one mapping between the elements in N_P and the predicates in P, and
(b) there is a one-one mapping between the elements in N_D and the variables definitions in P.

Any node D ε N_D of a GPG can be thought of as representing a 2-tuple, $<d,e_d>$, where d corresponds to a variable definition and e_d refers to the expression associated with this definition d. Each node Q ε N_P can be regarded as having only an expression, e_q, associated with it.

The edges in a GPG are also divided into two disjoint sets, called control edges and dependency edges. More specifically,

$$E = E_C \cup E_D,$$

where E_C refers to a set of control and E_D denotes a set of dependency edges, as defined below.

(a) $<u,\partial>$ ε E_C iff u, ∂ ε N and the code (definition or predicate) in P denoted by ∂ can be executed immediately after the execution of the code denoted by u in P. It is assumed that all control flow is explicit -- exceptions are not addressed.

(b) $<q,\mu>$ ε E_D iff q ε N_D and μ ε N, and q denotes a definition in P that can reach μ and the

variable defined by q is referenced by the expression associated with the code in P represented by μ. A definition q of variable v can reach node μ if there is a control flow path from q to μ that is free of redefinition of v [9].

Note that, the control flow edges in E_C are essentially the same as the edges in a flow graph. A GPG is an expanded flow graph with a node corresponding to each variable definition and predicate rather than the flow graph strategy of nodes representing basic blocks. Additional edges denoting dependencies are also included in the GPG. The edges in E_D represent the possible dependencies of nodes in N on definitions in N_D.

2.2 Sources of Data Dependencies in a Program

A variable definition may depend on the values of several independent variables for various reasons. A dependency can be determined by evaluating an individual statement in the program. Every possible data dependency resulting from the execution of the statement (isolated from the rest of the program) is represented by an edge. Statements that generally cause a change in the values of variables include input statements, assignment statements and procedure calls.

Consider an assignment of the form,

$$A := h(y_1, y_2, ..., y_n), n \geq 1$$

where A, y_1, y_2, ..., y_n are variables and h denotes some combination of operations on y_1, y_2, ..., y_n. There are n dependencies in this statement, since the assignment to A depends on the values of y_1, y_2, ..., y_n (assuming only one definition of each variable y_i, $1 \leq i \leq n$, can reach the statement).

The actual data dependencies resulting from an external procedure call can not be determined. Hence, all possible data dependencies that may result from an external procedure call are included in the GPG. The data dependencies resulting from an external procedure call statement depend on the modifiability of the procedure arguments. In particular, if the parameters are passed by reference and not protected from modification by the called procedure, then each argument may be affected by the values of the arguments. Consider a statement denoting procedure call,

CALL Q(x, y, z).

Assuming that each argument may be modified by the procedure Q, Figure 1 shows the data dependencies for the external procedure call statement from the rest of the program.

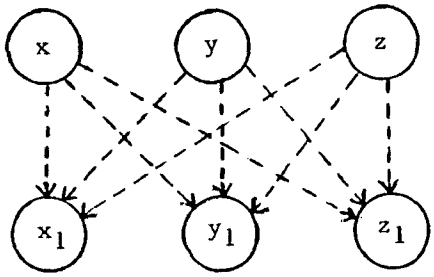

Figure 1. Possible dependencies for the procedure call statement

In Figure 1, x_1, y_1 and z_1 denote new definitions of x, y and z respectively. Also an edge <a,b> represents the dependency of the definition node b on a. One should note that, because definitions x_1, y_1 and z_1 are possible definitions and not absolute, x, y and z can still be referenced after the procedure call. A more detailed discussion about procedure calls is presented in section 3.

Global variables that may be referenced or modified by a called procedure are implicit arguments of a procedure call and hence result in additional data dependencies. In order to include the influence of global variables in the model, global variables referenced or modified by the procedure are treated as if they are explicit arguments of the procedure call.

2.3. Live Definitions of Variables

A variable definition is considered live at a statement, say S, if the value assigned may still be present on execution of S. Exact knowledge of live definitions at each statement in a program is required for constructing a GPG. In other words, one needs to know all the definitions of a variable X that can reach a variable definition d in order to correctly use definitions of X as the source node at d if d depends on X.

There are three control constructs, namely sequential code, branches and joins, that must be considered in determining live definitions of variables at any particular statement.

A variable definition for X kills all definitions that were live for the straight line code just prior to the new definition. However, a previous definition for X may remain live when the new definition is a possible definition such as a procedure call parameter. Since a procedure parameter might not be modified, all the earlier definitions for the parameter remain live. At a branch point all live definitions remain live until they are redefined. The live definitions after a join point consist of the union of all the definitions that were live on each predecessor of the join. The effects of branches and joins can further be illustrated using specific examples.

Consider the following IF-THEN-ELSE construct:

```
IF A > B THEN
    C := A
ELSE
    C := B ;
X := C ;
```

Note that, assignment to X depends on the variable C. Since C can receive its value in either the IF-THEN branch or the IF-ELSE branch, X is dependent on both assignment to C. The GPG representation of the above code segment is shown in Figure 2, where C_1 and C_2 denote two different definitions of variable C. p refers to the predicate node and both A and B are assumed to be initialized. Also, control edges are represented by solid lines and dotted arcs denote data dependencies.

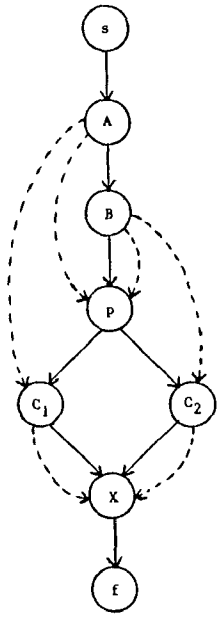

Figure 2. GPG Structure for IF-THEN-ELSE

Loops are considered as special cases of branches and joins. The loop exit is a branch, where one successor being the loop entrance and the other successor being the code following the loop. The loop entrance is a join, with one predecessor is the code for the first entrance and the other predecessor is the loop exit.

Consider the program segment containing a loop control structure as shown below.

```
X :=Y ;
WHILE X > Z DO
    X := X + Z ;
W := X * Z ;
```

In this case, the variable X used in the condition is considered to be dependent on both assignments to X since it is possible, at that point, that the value of X could have been set either before or within the body of the loop. Based on the same argument, it is evident that assignment to W also depends on both assignments to X. The GPG structure for this segment is shown in Figure 3, where X_1 and X_2 represent two different definitions of X. The other loop constructs can be treated analogously.

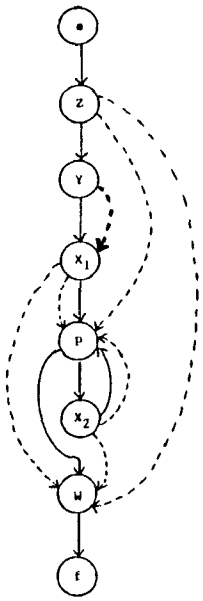

Figure 3. GPG structure for WHILE-DO

Finally assignment to an array element must be considered. Suppose the array, rather than its elements, is the data object of interest. Since an assignment to an array element modifies the contents of the array, the assignment kills the old

definition and creates a new definition. However, as the values of many of the array elements may remain unchanged after the assignment, the new definition is dependent on the old definition. Complex data objects other than an array would be treated in a similar manner.

2.4 Algorithm for Constructing a GPG

An algorithm for constructing a GPG is outlined below. The input to the algorithm is an imperative language program and the output is a GPG representation of the input program. For simplicity, it is assumed that all procedures are external and so the actual data dependencies of called procedures are not known.

Algorithm. CONSTRUCT-GPG(P).

Input. A source program, P, written in an imperative language.

Output. A GPG = (N, E, s, f).

Method.

A. Find the node set N consisting of all the variable definitions and the predicates in the program, P.

B. Draw a control flow graph,

$$G = (N, E_C, s, f),$$

where the edges in E_C describes the order of execution of the definitions in N within the program.

C. Add data dependency edges on the structure obtained in step B using the following rules.

C.1. For each statement S in the program P find the set of definitions that are live before execution of S, and the set of definitions that are live after execution of S. An algorithm for determining the live definitions of variables at each statement of a program can be found in [9].

C.2. For each definition in G use the appropriate routine below.

C.2.1. **Scalar Assignment**
$x := h(y_1, y_2, ..., y_n)$

(a) Find the set of live definitions, Ly_i, corresponding to each variable y_i for $1 \le i \le n$.

(b) For all definitions $d \in \bigcup_{i=1}^{n} Ly_i$, add the data dependency edges (d,x).

C.2.2. **Array Assignment**
$x(y_1, y_2, ..., y_m) := h(y_{m+1}, y_{m+2}, ..., y_n)$

(a) Use case (C.2.1) above to process as if the assignment were $x := h(x, y_1, y_2, ..., y_n)$.

C.2.3. **Procedure Call**
CALL $Q(x_1, x_2, ..., x_m, y_1, y_2, ..., y_n)$
where $x_1, x_2, ..., x_m$ are input arguments, and $y_1, y_2, ..., y_n$ are output arguments.

(a) Find the set of live definitions Lx_j corresponding to each x_j with $1 \le j \le m$.

(b) For each y_k, $1 \le k \le n$, for all $z \in \bigcup_{j=1}^{m} Lx_j$, add dependency edges (z,y_k).

C.2.4. **Predicates** (IF,WHILE, FOR, etc.)
For a given predicate node P_k use the following rules.

(a) Corresponding to each variable x_i in the conditional expression, find the set of live definitions Lx_i, $i \ge 1$.

(b) For all elements $w \in \bigcup_{i} Lx_i$, add the dependency edges (w, P_k).

D. The graph obtained using the above steps is a GPG = (N, E, s, f) corresponding to the input program P.

2.5. GPG Construction

We construct the generalized program graph for two program segments written in PASCAL. For the readers benefit, each definition node is labeled with a name identifying the variable that the node represents and a subscript. The subscripts are used to distinguish between nodes denoting different definitions of the same variable and are sequentially numbered based on

the relative position of the definition in the source code. The predicate nodes are not designed by any special labels. The expression associated with a predicate or a definition node is also implicit in the graph.

In the GPG diagrams, solid lines are used to denote control edges and dotted lines represent dependency edges. Explicit structures of the GPG's corresponding to two program segments are shown in Figure 4 and Figure 5.

Program 1.

```
0  Readln (A, B) ;
1  X := A ;
2  Y := B ;
3  While X < > B do
                Begin
4               X := X - 1 ;
5               Y := Y + X ;
                End ;
6  Z := X + Y + 2 :
7  Writeln (Z) ;
```

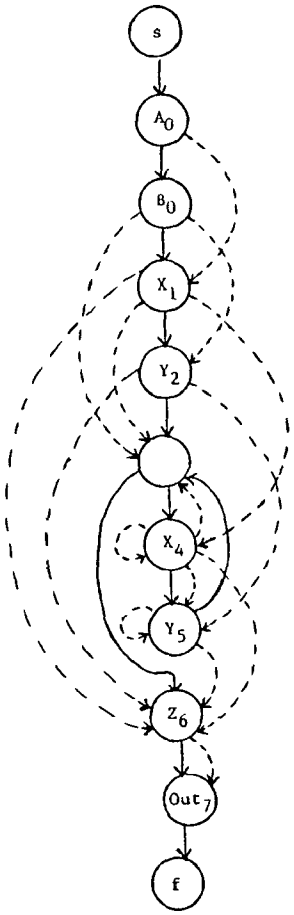

Figure 4. GPG Structure for Program 1

Program 2.

```
1           M := 5 ;
2           X := 3 ;
3           For I := 1 to M do
4               If X = L[I] then
5                   Go to 1 ;
6           L[I] := X ;
7           C[I] := 0 ;
8           M := I ;
9       1 : C[I] := C[I] + 1 ;
```

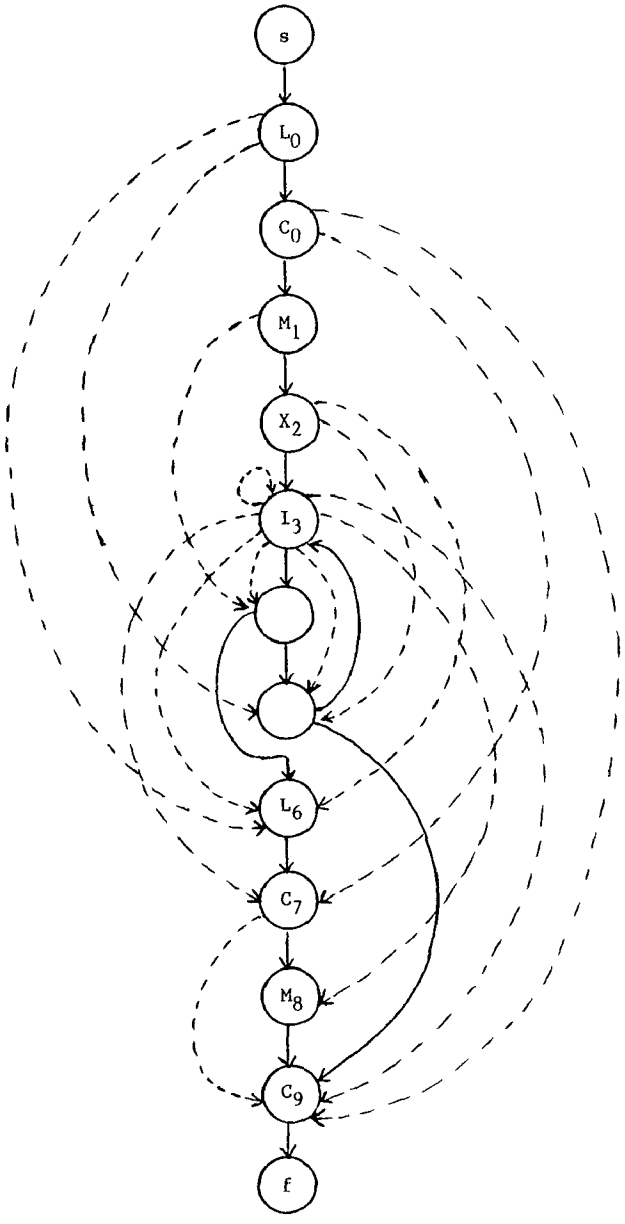

Figure 5. GPG Structure for Program 2

In figure 4, the nodes labeled A_0 and B_0 represent the definitions of the initial values of the variables A and B respectively, and these initial values are set by the input statement. The nodes X_1 and Y_2 denote the initial definitions of the corresponding variables at the appropriate positions of the source code. The node following the definition Y_2 refers to the condition of the WHILE loop. X_4 and Y_5 are the redefinitions of the variables X and Y at statement 4 and statement 5 respectively. Similarly, the node Z_6 refers to the definition of Z at statement 6. The Writeln statement is considered to be a definition of an output file, and is represented by the node labeled out_7.

The possible dependency edges were constructed based on the strategy described in Section 2.4. In statement 1, the assignment to X depends on the variable A. Since only one definition of A, namely A_0, is live at this statement, the edge $<A_0, X> \varepsilon E_D$. Clearly from statement 2, the edge $<B_0,Y> \varepsilon E_D$. The predicate at statement 3 is dependent on the variables X and B. A simple analysis shows that both the definitions of X (given by statements 1 and 4) and one definition of B can be live at statement 3, and thus the node representing the predicate has three dependency edges as drawn from X_1, X_4 and B_0. Data dependencies at other nodes were drawn following Algorithm 1, and are explicitly shown in Figure 4. Note that, the self loops at the nodes labeled X_4 and Y_5 mean that these definitions are dependent on themselves (from previous iterations).

Referring to Figure 5, one should note that the nodes labeled L_0 and C_0 denote definitions of the initial values of the arrays L and C respectively. We assume these initial values are set via explicit initialization or from input statements. The definition I_3 is followed by two predicate nodes -- one refers to the condition of the FOR loop and the other represents the condition of the IF statement. The dependency edges at nodes L_6, C_7 and C_9 are drawn based on the rule (C.2.2) of Algorithm 1.

3. STUDY OF SOFTWARE MODULE INTERCONNECTION

A software system typically consists of a set of interconnected modules. Therefore, the overall programmer/program complexity of a software depends on two major factors:
(1) intramodular complexity -- the complexity of the code within the modules and
(2) intermodular complexity -- the complexity due to interconnection between the modules. Most of the existing software measures focus on the complexity of an individual module and barely address the intermodular complexity.

In this section, we describe software module interfaces using the GPG model of module structure as a base. The GPG of individual module is reduced to eliminate information not relevant to interconnection. The individual reduced GPG's are then connected using the possible correspondence between actual and formal parameters. The resulting interconnection structure can be a very useful model to rigorously develop measures of interconnection complexity for a software system.

Given a GPG = (N, E, s, f) for a module i. a reduced GPG (RGPG) is a digraph

$$RGPG = (N_i, E_i, s, f),$$

where N_i is the set of nodes and $N_i \subseteq N$, E_i is a set of edges in $N_i \times N_i$, and $s, f \varepsilon N_i$. Moreover, N_i can be grouped into subsets,

$$N_i = N_{i,r} \cup N_{i,c} \cup N_{i,in} \cup N_{i,out} \cup s \cup f,$$

where $N_{i,r}$ = the set of definitions $r \varepsilon N$ such that r can reach a procedure/function invocation that references the variable defined by r as an argument,

$N_{i,c}$ = the set of definitions $c \varepsilon N$ which are created due to the invocation of a module j by the module i,

$N_{i,in}$ = the set of formal parameter definitions of module i, and

$N_{i,out}$ = the set of definitions corresponding to the variables or formal parameters whose values are returned to some calling module k within the system.

The nonempty set E_i is also divided into two disjoint sets $E_{i,C}$, a set of control edges, and $E_{i,D}$, a set of dependency edges, where

$E_i = E_{i,C} \cup E_{i,D}$ and

(a) $<u,\partial> \varepsilon E_{i,C}$ iff $u, \partial \varepsilon N_i$ and either $<u,\partial> \varepsilon E_C$ or there exists a control flow path $u, x_1, x_2,$

..., x_n, ∂ in the GPG such that each $x_j \in (N-N_i)$, $1 \le j \le n$.

(b) $\langle x,y \rangle \in E_{i,D}$ iff $x \in N_i$, $y \in (N_{i,in} \cup N_{i,out} \cup N_{i,r})$, and either $\langle x,y \rangle \in E_D$ or there exists a dependency path $x, w_1, w_2, ..., w_m, y$ in the GPG with each $w_k \in (N-N_i)$, $1 \le k \le m$.

Module interfaces consist of the flow of control between modules and the dependencies between parameter definitions and references. The interconnection structure that we define includes the set of RGPGs in a system and a set of interaction edges that connect nodes in different RGPGs.

It is assumed that communication between a calling and a called module is maintained only via parameter passing. Global variables are treated as implicit parameters. We also assume that whenever a module is invoked by another module the execution starts at the start node s and terminates at the exit node f. Interface between any pair of RGPGs is represented by interaction edges which connect the nodes denoting the definitions of actual parameters with the nodes representing the definitions of formal parameters.

Formally, the structure of software module interconnection is defined to be a 2-tuple, IG = $\langle G,I \rangle$, where

(a) G is a finite set of RGPG's, $G = \{G_1, G_2, ..., G_n\}$, such that $G_i = (N_i, E_i, s_i, f_i)$ is the RGPG for module i, $1 \le i \le n$, and n is the number of modules in the system.

(b) I is a set of directed interaction edges connecting the parameter definition nodes in different RGPGs,

$$I = \bigcup_{m=1}^{M} \left(\bigcup_{k=1}^{N_m} E_{j_m}^{k} \right)$$

where $j_1, j_2, ..., j_M$ are the calling modules in the system, N_m is the number of modules invoked by a particular modules j_m, $1 \le m \le M$, and $E_{j_m}^{k}$ is the set of interaction edges between the calling module j_m and the k^{th} module invoked by j_m,

$E_{j_m}^{k} = \{ \langle x,y \rangle \mid (x \in N_{j_m,r}$ is bound to the corresponding $y \in N_{k,in}$) or ($x \in N_{k,out}$ is bound to $y \in N_{j_m,c}$)$\}$.

Algorithms for producing a RGPG from a given GPG and for constructing the interaction edges of a system, and a detailed analysis of the interconnection structure are presented in [8]. The analysis is directed towards the development of useful measures of software module interconnection structure.

Other theoretical and practical applications of the GPG are quite apparent. The GPG can conveniently be used as an analytical tool for studying software structure. Some preliminary results of the analysis of the GPG structure and the interface between control and data dependencies are reported in [7]. As our work moves from an analytical to an empirical perspective, we see the GPG as an ideal implementation model for the development of software structure and measurement tools. We can construct a set of software tools based on the GPG to analyze control structure, data dependency structure, and the control and data dependency interface. Software tools can be designed to compute proposed measures from arbitrary GPG's. Since the structural content of the original source is more completely captured by its GPG representation than by the control flow graph or the data dependency graph independently, most of the new structural abstractions and measures can directly be applied to the GPG representation.

There are pragmatic advantages of basing software structure and measurement tools on an abstract representation rather than the actual source code. Researchers can develop one set of tools that analyze the abstract representation. These tools can be used to study programs written in any language for which a GPG construction program has been implemented. Only the GPG construction program would have to be implemented individually for each language. Industry has been reluctant to release programs to software structure and measures researchers because of proprietary and copyright concerns. Proposals for developing a standardized reduced form of a program for software complexity research purposes have recently appeared in the literature [10]. We feel that the GPG is a suitable candidate. Industry should be less reluctant to share valuable data when the software is converted into its abstract

GPG representation before release to the research community.

4. CONCLUSIONS

A general abstraction, called generalized program graph, for representing arbitrary imperative programs is developed in this paper. Some important aspects of GPG construction were explicitly discussed and an algorithm for building the GPG was presented. The GPG can prove useful for examining the static structural relationship between control flow and data dependencies. Moreover, applications of the GPG in studies of module interconnection complexity appears to be encouraging. It is expected that the directed graph structure of the software module interconnection, incorporating the effect of both control flow and data dependencies, will aid in the effort to isolate quantifiable software properties that can be examined in a rigorous manner. Finally, the GPG may form a basis of an implementation model for the development of software structure and measurement tools.

References

[1] McCabe, T.J., "A Complexity Measure", *IEEE Tran. Software Engineering*, Vol. SE-2, December 1976, pp. 308-320.

[2] Oviedo, E. I., "Control Flow, Data Flow and Program Complexity", *Proceedings COMPSAC 1980*, pp. 146-152.

[3] Harrison, W. A. and Magel, K. I., "A Complexity Measure Based on Nesting Level", *ACM SIGPLAN Notices*, Vol. 16. No. 3, March 1981, pp. 63-74.

[4] Evangelist, M., "An Analysis of Control Flow Complexity", *Proceedings COMPSAC 1984*, pp. 388-396.

[5] Howatt, J. W. and Baker, A.L., A New Perspective on Measuring Control Flow Complexity, *Technical Report 85-1*, Computer Science Department, Iowa State University, 1985.

[6] Bieman, J. M., *Measuring Software Data Dependency Complexity*, Ph.D. Dissertation, University of Louisiana, 1984.

[7] Debnath, N. C. and Bieman, J. M., An Analysis of Software Structure Using. a Generalized Program Graph, *Technical Report 85-7*, Computer Science Department, Iowa State University, 1985.

[8] Bieman, J. M. and Debnath, N. C., The Structure of Software Module Interconnection, *Technical Report 86-2*, Computer Science Department, Iowa State University, 1986.

[9] Hecht, M.S., *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, 1977.

[10] Harrison, W. and Cook, C., "A Method of Sharing Industrial Software Complexity Data", *ACM SIGPLAN Notices*, Vol. 20, No. 2, February 1985, pp. 42-51.