COMPSAC '85

AN ANALYSIS OF SOFTWARE STRUCTURE USING A GENERALIZED PROGRAM GRAPH

James M. Bieman and Narayan C. Debnath

Department of Computer Science
Iowa State University
Ames, Iowa 50011

Abstract

A generalized program graph (GPG) is an abstraction of a program where nodes denote variable definitions and predicates, and edges denote either control flow or data dependencies. A GPG can be constructed from programs written in arbitrary imperative programming languages. Analyses of GPG representations show the interconnection between control flow constructs and data dependencies. Using simple transformations, a GPG can be converted into a control flow graph or a data dependency graph and can therefore be used to study control and data dependency issues in isolation. Because the GPG includes both the control and dependency information in one abstraction, the GPG 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 (CFG) has been used to study structured programming, testing methodologies, and develop complexity measures [McCabe 76, Oveido 80, Harrison 81, Evangelist 84, Howatt 85]. The data dependency graph (DDG) 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 [Bieman 84, Bieman 85]. 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 CFG alone.

In order to study the interface between control structure and data dependencies, we use a generalized program graph (GPG). The GPG incorporates features of both the CFG and the DDG. The GPG representation is essentially equivalent to the string form of a program. However, because of its graph structure, a GPG is better suited for structural analysis than the original string form.

In this paper, we define the GPG and show some examples of GPG construction. We present the initial results of our analysis of the GPG structure and the interface between control and data dependency complexity. Finally, we discuss some obvious practical and theoretical applications of the GPG.

2. Generalized Program Graph

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 determined by control flow constructs. A variable definition consists of the name of the variable to be redefined, an assignment operator and an expression that evaluates to the value assigned in the definition. 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, and this boolean result determines the next statement to be executed (we assume only binary predicates).

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. Moreover, the graph has an unique entry node, s, and an unique exit node,

0730-3157/85/0000/0254$01.00 © 1985 IEEE

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

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

with a finite set of nodes, $N \cup \{s,f\}$, and a finite set of edges, E. In particular,

$$N = N_P \cup N_D$$

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 variable definitions in P.

Any node $D \in N_D$ of a GPG can be thought of as representing a 2-tuple, $\langle d,e_d \rangle$, where d corresponds to a variable definition and e_d refers to the expression associated with this definition, d. Each node $Q \in N_P$ of a GPG 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 edges and E_D denotes a set of dependency edges, as defined below. We are assuming that all control flow is explicit - exceptions are not addressed.

(a) $(u,v) \in E_C$ iff $u, v \in N$ and the statement denoted by v in P can be executed immediately after the execution of the statement denoted by u in P.
(b) $(q,r) \in E_D$ iff $q \in N_D$ and $r \in N$, and q denotes a definition in P that can reach r and the variable defined in q is referenced by the expression associated with statement represented by r in P. A definition of variable v can reach node r if there is a control flow path from q to r that is free of redefinition of v [Hecht 77].

Note that the control flow edges in E_C are essentially the same as the edges in a flow graph. One can view a GPG as an expanded flow graph with a node corresponding to each variable definition and predicate rather than only for basic blocks. Furthermore, edges denoting dependencies are added to the GPG. The edges in E_D simply represents the functionality or dependency of nodes in N on definitions in N_D.

3. GPG Construction

In this section we construct the generalized program graph for two program segments written in PASCAL. As defined in the previous section, each node in a GPG denotes either a predicate or a variable definition. A variable definition is generated at each statement that may modify the value of a variable. Such statements include assignment statements and input statements. Initialization of a variable is also considered to be a definition. 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 designated 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. In order to draw dependency edges in the GPG, one must determine which definitions are live 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 a dependency edge, 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. Explicit structures of the GPGs corresponding to two program segments written in PASCAL are shown in Figure 1 and Figure 2.

In Figure 1, the node labeled Lrg_1 denotes the definition of the initial value of the variable Lrg. The node labeled New_3 represents the definition of initial and successive values, if any, of the variable New. Similarly, the node labeled Lrg_5 is the redefinition of variable Lrg at statement 5, and 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 discussed in [Bieman 84]. For example, in statement 4, the final value of the predicate is dependent upon the definitions of variables Lrg and New that may possibly reach this statement. Since both the definitions of Lrg, given by statement 1 and statement 5, and only one definition of New can be live at statement 4, the node representing the predicate at this statement has three dependency edges as drawn from nodes labeled Lrg_1, Lrg_5, and New_3. The redefinition of Lrg at statement 5 and the node denoting the condition of the repeat..until loop at statement 6, each has one dependency edge corresponding to only possible live definition of the variable New given by statement 3. Finally, two possible live definitions of variable Lrg at statement 7 provides two dependency edges drawn at the node labeled Out_7.

Referring to Figure 2, one should note that the nodes labeled A_0 and B_0 represent the definitions of the initial values of the variables A and B, respectively. We assume these initial values are set via explicit initialization or from input

Program 1.

```
1      Lrg := 0;
2      Repeat
3          Readln(New);
4          If Lrg < New then
5              Lrg := New
6      Until New < 0;
7      Writeln(Lrg);
```

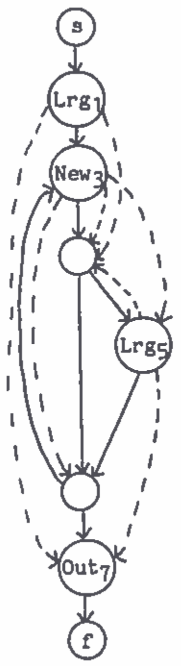

Figure 1. GPG for Program 1

Program 2.

```
1      X := A;
2      Y := B;
3      Z := 0;
4      While Y <> 0 DO
           Begin
5              Y := Y - 1
6              Z := Z + X
           End;
7      W := Z + 2;
```

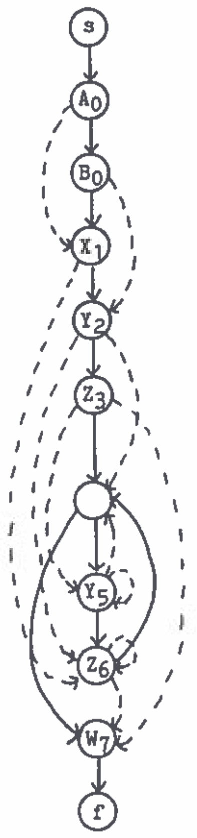

Figure 2. GPG for Program 2

statements. The nodes X_1, Y_2, and Z_3 denote the initial definitions of the corresponding variables at the appropriate positions of the source code. The node following the definition Z_3 refers to the condition of the while-loop. The redefinitions of Y and Z are represented by the nodes labeled Y_5 and Z_6, respectively. The node W_7 shows the only definition of the variable W. All the dependency edges at various nodes of the graph were drawn in the same way as in Figure 1. Note that, the self loops at the nodes labeled Y_5 and Z_6 mean that these definitions are dependent on themselves (from previous iterations). A detailed discussion about the various sources of dependencies and how to address specific constructs such as arrays, functions, and procedure calls are given in [Bieman 84].

4. Some Basic Properties Of The GPG

We present some basic properties of generalized program graph and associated definitions. Because the following properties directly follow from the construction of the graph, they are described as properties with informal justification rather than as formal theorems.

Property 1. For every dependency edge (d,p) in a GPG, $d \in N_D$ and $p \in N$, there is a control flow path from d to p.

The dependency edge (d,p) can exist only if the definition represented by d can reach p by a control flow path.

Definition 1. A path $v_1, v_2, \ldots, v_k$ with $(v_i, v_{i+1}) \in E_D$ and $i=1,\ldots,k-1$, is defined to be a dependency path from v_1 to v_k.

Property 2. For every dependency path in a GPG that is a cycle, there is a control flow cycle in the graph. This follows directly from Property 1.

Therefore, the existence of a control flow cycle is necessary for a dependency self loop.

Definition 2. The Range of a predicate p is the set of nodes that fall on any path from the immediate successors of p to the first node where all paths from p merge, the GLB(p). The GLB(p) is not included in the Range of p. [Howatt 85].

Definition 3. A definition d of a variable v is considered live at node $n \in N$ if d can reach n.

Definition 4. For a predicate $p \in N_P$ and a variable v, INFLUENCE (p,v) is defined to be the set of nodes of the GPG at which any definition of v made within the range of p are live.

Property 3. INFLUENCE (p,v) in a GPG may extend beyond the range.

To illustrate this property, consider the INFLUENCE of the predicate given in the code below and the associated GPG shown in Figure 3.

code

```
0 : A := input value
    if p then
       X;
1 :    A := f(A)
    else
       Y;
2 :    A := g(A)
    endif
    Z := f(A)
```

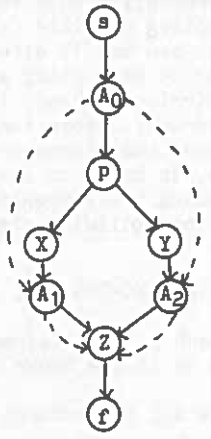

Figure 3. The Influence may extend beyond the Range

In Figure 3:

$A, X, Y, Z \in N_D$ and $p \in N_P$,

Range of the predicate p = $\{X, Y, A_1, A_2\}$,

GLB(p) = Z,

INFLUENCE (p,A) = Z, and

Z $\notin$ Range of p.

Suppose A_0 is the initial value of the variable A, and A_1, A_2 are the redefinitions of A made within the range of the predicate shown. Assuming that Z is defined as a function of A, it is clear that both the definitions A_1 and A_2 are live at node labeled Z. Also, the node labeled Z is the GLB(p) and hence Z does not belong to the range of this predicate. This result is purely due to dependency edges that are added in the graph, and the property is not evident from the control flow graph alone.

Property 4. INFLUENCE (p,v), $p \in N_P$ and v is a variable, ends at a point past GLB(p) where v is redefined in all paths from GLB(p).

The redefinition of the variable v at node n kills any previous definitions of v that reach n.

Property 5. For a given predicate, p, to maximize the increase in number of live definitions of a variable, v, there must be one and only one path from p to the GLP(p) free of definition of v.

Consider the graph segments shown in Figure 4(a) and Figure 4(b).

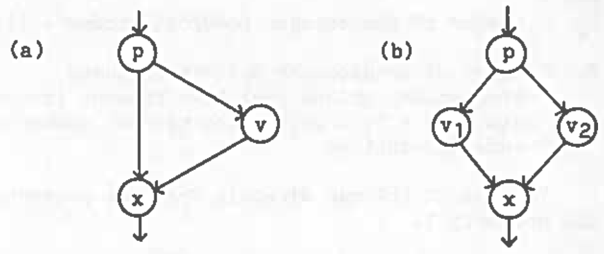

Figure 4. A Predicate Maximizing the Increase in Live Definitions

Assume that the number of live definitions of variable v at the predicate node is n. Then in case (a), the number of live definitions of v at node labeled x is (n+1). While in case (b), the number of live definitions of v at node labeled x is only 2, irrespective of the value of n, since the redefinitions of v in both the paths kill all earlier definitions of v.

Property 6. For each node $n \in N$ and for each variable v referenced at n, the maximum number of live definitions of v at n is equal to the number of predecessor predicate nodes + 1.

Clearly, predicates can increase the number of definitions that may be live when control exits the range of a predicate. In general, for each variable v, a predicate p can increase the number of definitions of v that can reach GLB(p) by one (assuming a 2 alternative predicate). The result follows directly for any sequential or nested control flow constructs.

Property 7. For each node $n \in N$, the maximum number of referenced variables is the number of definition nodes, $d \in N_D$, representing the definitions of unique variables that are predecessors of n on a path from s to n containing the maximum number of definitions of unique variables.

There can be a number of paths from s to a given node $n \in N$. Each path might contain a different number of unique variable definitions that are predecessors of n, and these definitions can also be referenced at the node labeled n. Choose the path, from s to n, that contains maximum number of definitions of unique variables. Trivially, the number of unique definitions in this path sets the upper limit for the number of variable definitions which can be referenced at the given node n.

Property 8. The maximum indegree of dependency edges for any node $n \in N$ in a GPG is given by the expression:

$$P_p * P_v,$$

where

$P_p \equiv$ (number of predecessor predicate nodes + 1), and

$P_v \equiv$ number of predecessor definition nodes representing unique variables present in the path from s to n containing highest number of variables defined

The result follows directly from the property 6 and property 7.

Property 9. The maximum dependency outdegree of any node $n \in N_D$ is the number of successor nodes $m \in N$ along the control flow edges.

A variable definition given at a node $n \in N_D$ can be referenced by any successor node $m \in N$. As a result, there can be a dependency edge from n to any m if (i) the definition at n is live at node m (ii) there exists a control flow path from n to m and (iii) the expression associated with the node m is a function of the variable set by the definition associated with the node n. It follows that the maximum dependency outdegree for any node $n \in N_D$ is achieved when evaluation of all the expressions associated with successor nodes $m \in N$ depends on the definition at $n \in N_D$.

Property 10. The maximum number of dependency edges in a GPG is given by the

expression:

$$\sum_{\text{for each } m \in N} \text{(maximum indegrees of dependency edges)}$$

The maximum indegree of dependency edges for any node $m \in N$ can be obtained by property 8. This formula sets an upper limit on the number of dependency edges based on the control flow.

5. Transformation Of A GPG Into CFG And DDG

Since both the control flow as well as data dependency information are included in the GPG, straightforward transformations can be used to convert a GPG representation of a program into a control flow graph and/or a data dependency graph. The conversion demonstrates that the GPG can be used to generate complexity measures based on the CFG and the DDG.

5.1. Conversion of a GPG into a CFG

In order to transform a GPG into a CFG, the following steps are performed.

(1) Delete all the dependency edges denoted by dotted lines.

(2) Combine each set of nodes that represents a basic block into a single node.

In a flow graph, each node generally represents a basic block. However, after step 1 of the above transformation, one obtains a control flow graph where each node refers to an individual statement of the source program. It is therefore required to condense each appropriate set of nodes into a single node, so that after step 2 of the transformation the given GPG is transformed into a standard control flow graph.

5.2. Conversion of a GPG into a DDG

A DDG representation of a program consists of nodes representing only variable definitions and edges denoting possible data dependencies. A dependency can be (1) direct dependency that can be determined by evaluating an individual statement or (2) a control dependency that is obtained from the flow of control. Note that direct dependencies have already been incorporated into the GPG. Therefore, in order to convert a GPG into a corresponding DDG representation of a given program, the following algorithm can be used.

```
Procedure GPG-TO-DDG (G);

    For each $p \in N_P$, determine the set $\{d \mid d \in N_D$
    and d is in the Range of p$\}$;

    Delete all the control flow edges in the graph,
    G;

       For each predicate $p \in N_P$
          Do;
          Find the dependency edges $(q_i,p)$ of p,
          for $q_i \in N_D$;

       For each $q_i$, add the edge $(q_i,d)$ for all
       $d \in N_D$ that lie within the range of p;

       Delete the predicate node p together with
       all the edges $(q_i,p)$ of p
       for $q_i \in N_D$;
       End;

    The resulting graph, G', is a DDG.
```

6. Applications Of The GPG

Our current work has centered on the use of the GPG as an analytical tool for studying software structure. 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. The GPG can be used to represent programs written in imperative languages. We can construct a set of software tools based on the GPG to analyze control structure, data dependency structure, and the control/dependency interface. We also expect to find applications of the GPG in studies of interconnection complexity. 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 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 [Harrison 85]. 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.

7. Conclusions

The results presented in this paper represent the state of our ongoing research examining the static structural relationship between control flow and data dependencies. The generalized program graph is an abstraction that has proven useful for the analysis. In addition, the GPG may form a basis of an implementation model for the development of software structure and measurement tools.

We have not proposed any new software measures. But we feel that the GPG will aid in the effort to isolate quantifiable software properties that can be examined in a rigorous manner.

References

[Bieman 84] J.M. Bieman, Measuring Software Data Dependency Complexity, Ph.D. Dissertation, Computer Science Dept., Univ. of Louisiana, Lafayette, Louisiana, 1984.

[Bieman 85] J.M. Bieman and W. Edwards, "Experimental Evaluation of the Data Dependency Graph For Use in Measuring Software Clarity", Proc. of the 18th Hawaii International Conference on Systems Sciences, 1985, pp. 271-276.

[Evangelist 84] M. Evangelist, "An analysis of control flow complexity", COMPSAC 1984, pp. 388-396.

[Harrison 81] W.A. Harrison and K.I. Magel, "A complexity measure based on nesting level", ACM SIGPLAN Notices, Vol. 16, No. 3, March 1981, pp. 63-74.

[Harrison 85] W. Harrison and C. Cook, "A Method of Sharing Industrial Software Complexity Data", ACM SIGPLAN Notices, Vol. 20, No. 2, February 1985, pp. 42-51.

[Hecht 77] M.S. Hecht, Flow Analysis of Computer Programs, Elsevier North-Holland, New York, 1977.

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

[McCabe 76] T.J. McCabe, "A Complexity Measure", IEEE Trans. Software Engineering, Vol. SE-2, December 1976, pp. 308-320.

[Oviedo 80] E.I. Oviedo, "Control flow, data flow and program complexity", COMPSAC 1980, pp. 146-152.

[Weiser 82] M. Weiser, "Programmers Use Slices When Debugging", Communications of the ACM, Vol. 25, No. 7, July 1982, pp. 446-452.

[Woodward 79] M. Woodward, M. Hennell, and D. Hedley, "A measure of control flow complexity in program text", IEEE Trans. Software Engineering, Vol. SE-5, January 1979, pp. 45-50.