THE PROGRAM DEPENDENCE GRAPH IN A SOFTWARE DEVELOPMENT ENVIRONMENT

Karl J. Ottenstein
Linda M. Ottenstein

Dept. of Mathematical and Computer Sciences
Michigan Technological University
Houghton, Michigan 49931

ABSTRACT

The internal program representation chosen for a software development environment plays a critical role in the nature of that environment. A form should facilitate implementation and contribute to the responsiveness of the environment to the user. The program dependence graph (PDG) may be a suitable internal form. It allows programs to be sliced in linear time for debugging and for use by language-directed editors. The slices obtained are more accurate than those obtained with existing methods because I/O is accounted for correctly and irrelevant statements on multi-statement lines are not displayed. The PDG may be interpreted in a data driven fashion or may have highly optimized (including vectorized) code produced from it. It is amenable to incremental data flow analysis, improving response time to the user in an interactive environment and facilitating debugging through data flow anomaly detection. It may also offer a good basis for software complexity metrics, adding to the completeness of an environment based on it.

Categories and Subject Descriptors: D.1.2, D.2.2, D.2.5, D.2.6, D.2.8, D.3.4

Additional Key Words and Phrases: internal program representation, data flow, control flow, code optimization, program slice, interpreter, debugging, software complexity metrics

Partial support for the authors was provided by the National Science Foundation under Grant No. MCS-8203487.

1. Introduction

A software development environment is generally conceived of as an interactive system which supports program development in an integrated fashion [18]. The editors, compilers, linking loader, word processors, program data base system, run-time system, interactive debugger and other tools provide more assistance to the programmer in producing correct code quickly when they operate in a consistent framework that is knowledgeable about the syntax and semantics of the source languages and, if possible, knowledgeable about the problem being solved.

The internal program representation chosen for a software development environment plays a critical role in the nature of that environment. The internal form is a major part of the nucleus of the system design, affecting potentially all tools. A form should facilitate implementation by permitting all environmental operations to be performed by easily programmed procedures. Further, the algorithms employed by those procedures should be fast in order to support the interactive nature of the environment. It is particularly important to note that the "environmental operations" under consideration consist of more than support for translation and editing functions. An internal representation should support debugging tools and the gathering of software complexity metrics as well.

Each internal representation is most appropriate in specific contexts. Abstract syntax trees, for example, are perfect if the only concern is editing and the generation of straightforward code. Syntax trees can capture the structure and content of the entire source program so that the intermediate can be the only program representation extant during program manipulation. (That is, a separate source file need not be retained.) Syntax trees do not represent any data flow information; however, and control information is not abstracted, as it mirrors the source. They are thus unsuitable for optimization and a variety of other tasks.

The program dependence graph (PDG) [13] has a number of advantages as an internal form in a software development environment. It can support editing, translation, debugging and program metrics. The PDG may be interpreted or it may have highly optimized (and vectorized [31]) code generated from it if desired [13]. It is thus suitable for an environment supporting both execution models. In addition, the PDG is amenable to incremental data flow update, an important feature in an interactive environment in which it is helpful to report data flow anomalies as they occur in order to aid in debugging [21]. Finally, the PDG appears useful for the gathering of software complexity metrics.

The remainder of this paper will address each of these issues in turn: the PDG, a debugging tool known as "slicing", execution models, incremental graph updating, software complexity metrics and practical considerations.

2. The Program Dependence Graph

It is necessary for the internal program representation in a software development environment to capture as much data and control dependence information as possible in order to effectively support the types of environmental operations suggested in the introduction. Much related work has been performed over the past nine years in this area of dependence-based program forms. Dennis' work [8] opened up the field of data flow computation [9]. Most representations in that area treat all dependences as data dependences, control dependences being converted as necessary. (Some forms, however, do treat control differently [30].) The program plans [33] of the Programmer's Apprentice project [32] represent control and data dependences in a modularized form in which loops have been converted to recursion. The goals there involve program understanding to aid modification. The data dependence graphs used in the Illinois vectorizer [24] are designed to hierarchically analyze dependence relations in programs. Further work involving the ideas of dependence depth and loop carried dependence [2,3] for vectorizing transformations has been carried out at Rice University. IF1 [27] is a proposed intermediate for applicative languages, but offers too compartmentalized a view to support major program restructuring. The data flow graph [23,22], conceived with program optimization in mind, represents global data dependence at the operator level. Transformations which involve both control and data dependence cannot be specified in a consistent manner with this form, however, since control is represented by a conventional control flow graph. The extended data flow graph [12] represents control dependence consistently with data dependence, but can only represent "structured" programs. The program dependence graph [13] eliminates this restriction on control flow. It is the abstraction of control dependence in these latter two forms which makes them so suitable for constructing program slices (discussed in the next section) and for performing major restructuring optimizations [12,13].

Figure 2(a) gives the usual control flow graph for the program fragment in Figure 1, annotated to indicate the locations of the blocks of program statements. Figure 2(b) gives the control dependence subgraph of the PDG for the same program. "Entry" is the condition for execution of the program, and is a distinguished node in every PDG. Predicates define regions, labeled Ri. All operators (statements) in a region are successors of a region node.

Note the difference in structure between these two graphs. The PDG subgraph shows that the two loops and final statement may be independent of each other. As far as control information is concerned, this may be the case. Data dependences might, and in this example will, actually impose a sequencing on these regions. The PDG subgraph does not show statement block S1 as being in a loop. From a control dependence standpoint, S1 is loop invariant: it is the data dependence cycle on I which determines that S1 is loop dependent. The implied organization given by the conventional control flow graph is limiting to optimization (because it hinders the rearrangement of large sections of code as done in [12]). It is also limiting to slicing, as we shall describe in the next section.

Figure 3 shows the PDG for this example. The solid edges represent data dependence and dashed edges represent control dependence. (We only illustrate the PDG through this example. The definition appears in [12] and allows for the various forms of dependence described in [24].) Figure 3 is not quite accurate in that there should be only one node for each program constant. This is relaxed here to increase the clarity of the figure. The update operator takes as input an array, an index and a value and produces an updated array as its value. Near the top of the figure, the initialization of the array a is performed with an update operator. The node a which is an input to the update represents the set of reaching definitions for the array a. Select retrieves the value of a given element from an array. The objects input and output represent file descriptors and are thus updated when operations are performed on them. (File descriptors are both inputs and outputs of most I/O procedures.) The two read operators used here would be merged into one operator in the actual PDG since they reside in the same predicate region. They remain distinct here to permit us to illustrate the importance of making the implicit input file explicit in the intermediate form as an aid to slicing.

3. Slicing with the PDG

Slicing is the abstraction of sets of statements which influence the value of a variable at a particular program location [35,36]. An experiment by Weiser [36] indicated that programmers use slices when debugging and could therefore benefit from the development of a tool to provide slicing information automatically. A potential environment using slicing could allow the user to edit a program using slice membership as the basis for inclusion in a window rather than syntactic structure alone. A debugger could display the offending slice in a window on an error condition or breakpoint rather than the entire syntactic context. (We assume a display routine which would provide automatic eliding of deeply nested statements, as is done in LISPEDIT [1,20]. Partial slices, without elision, are displayed for COBOL programs by [29].)

The computation of slices is based on data dependence as well as control dependence. A computation which affects the value of a variable at a desired observation point may be under the influence of a predicate (i.e., executed only when a predicate has a particular truth-value). Those statements which make up the control structure using the predicate must be included in the slice along with all statements in the data dependence subgraph which determine value flow to the observation point.

Explicit data and control dependence make the PDG ideal for constructing program slices. The PDG must be augmented with source text indicators such as (file, line pointer) pairs. The source text must be stored in an augmented form where each token points to the node in the PDG which represents it. (When common subexpression elimination is enabled, this mapping will not be one to one.) Some tokens, such as parentheses and block delimiters, which merely state precedence conditions will have no corresponding PDG node, but can be inferred from the PDG structure.

Since we are interested in walking back through the graph (and not forward), we can reverse the direction of the pointers in the physical PDG. Our pointers, therefore, tell us where data came from rather than where it is flowing. (This implementation is in fact sufficient for many optimization and code generation algorithms [22,23]; however, some optimizations as well as interpretation require forward flow information also, necessitating a doubly-linked implementation.)

How is a slice built? Suppose a user selects a variable, expression or statement in a program and wants the slice which is relevant to all variables in that object. The augmented source file points to the corresponding PDG nodes. A linear-time graph walk is made backwards from each of these nodes, building up a set containing the source text line indicators for all visited nodes. The walk terminates at either inputs, constants, or already-visited nodes. The slice set is then used to retrieve the relevant source lines for display. A source line may contain several statements, though, with only some being in the slice. The augmented source file allows us to delete these irrelevant statements since we can examine each token to see if its corresponding PDG node was visited in the walk.

Consider Figures 1 and 3. To construct the slice for a[last] in the final writeln, we start at the left and bottom-most select operator and walk backwards on all paths. Marking the immediate control predicates of included lines allows the determination of when to include precedence tokens such as begin-end. We thus obtain:

```
for i := 1 to last do
   a[i] := 0;
while not eof do
   begin
      read (code, value);
      read (dummy);
      if code = 'a' then
         a[value] := a[value] + 1
   end;
writeln (a[last], b[last]);
```

The representation of I/O with explicit file descriptors in the PDG caused the statement read (dummy) to be included in the slice. If it were not included, the programmer would be hard-pressed to find an input file sequencing bug. (All related I/O statements are not included in Weiser's slices.)

This slice could not have been as readily obtained were a conventional control flow graph part of the representation. Assume arbitrary control flow between the two loops in Figures 1 and 2(a). As long as no data values in that region affect the data values in our slice, none of that structure would ever be seen during our walk of the PDG. With a control flow graph, however, we would have to determine that none of the control-related computations on that path affected the computation. Weiser's algorithm for doing this is $O(n\,e\log e)$ where **n** is the number of nodes in the control flow graph and **e** is the number of edges. In debugging, one is apt to find the slicing tool most helpful when the portions of the slice are dispersed widely throughout the program. It is precisely in this situation that analyzing the control flow would be slow. But, traversing the PDG remains linear in time and is proportional to the amount of information in the slice.

This discussion on using the PDG to obtain slicing information has assumed that we are dealing with an untransformed PDG. In order to obtain slices from a PDG transformed by optimization algorithms, additional information must be posted to the graph. For example, when a common subexpression is eliminated by replacing a dependence edge from one computation with an edge from another, a marker node must be inserted along the new edge to retain the correct source line information. In other cases, nothing special need be done since some optimizations lead to more accurate slices. Invariant code motion causes loop independent computations to be moved outside of loops, maintaining safety by moving any guarding predicates as necessary [12]. If a slice includes a computation which is loop invariant, the loop control structure is irrelevant and should not be presented to the programmer.

4. Execution Models

The method of program execution chosen for an environment is important in determining an appropriate internal form. Some forms may be more suited to interpretation than the generation of object code or vice versa. The PDG is amenable to both execution models.

The PDG has been designed with the idea of producing highly optimized code for a variety of architectures, including parallel ones [13]. Zellweger's work deals with some of the problems of using optimization in a programming environment supporting an interactive debugger [37]. Hennessy addresses the difficulty of this issue [16].

A PDG can be interpreted, but not by conventional methods. It is a highly parallel form with no linear sequencing of operators. It is thus convenient to interpret it in a pseudo-parallel fashion which is both control and data driven. We maintain a list of nodes for which all data and control inputs are available. Nodes are removed from the list, interpreted, and the resulting value is transmitted to successors. Each successor is examined at once and is added to the execution list if it does not require any additional inputs. The virtual machine which our interpreter implements, then, is a form of data flow machine [9]. Unlike those machines, side-effects are not precluded here since, e.g., arrays may be implemented either functionally or using a memory model.

Every PDG or slice has the distinguished predicate _entry_ which must be "evaluated" before anything else in the graph may be. The execution list is thus initialized to contain the node _entry_, given the simple evaluation rule: _entry_ --> _true_. For consistency and efficiency, all values may be passed through the graph during interpretation by copying pointers. No separate storage allocation phase is required, as storage may be allocated and freed dynamically during interpretation.

5. Incremental Update

The internal representation could be constructed completely each time a user stops editing and wants to ask a question about the program or to execute it. In most situations, it might be preferable to update the internal form while the user is editing the source to minimize response time delays. Since the PDG represents all data and control flow information explicitly, this means that _incremental_ flow analysis techniques must be employed to avoid complete PDG reconstruction. (An incremental analysis attempts to propagate the new information without reexamining the entire program.)

Incremental analysis is needed when new definitions are added, old ones removed or the control flow changed. It is not required when new references to existing objects are added. But, to support the modification of the PDG in this case, it is necessary to retain the basic data flow information for each basic block. When the data flow analysis is performed to originally construct the PDG, we have information on all definitions which reach each statement. Only the information on definitions _used_ is retained in the resulting PDG. Yet, retaining those original sets of definitions allows us to immediately determine if there are any definitions for a new reference to a variable.

A new incremental method for updating data and control dependences when a branch is removed or a loop unrolled is sketched in [13] and will be detailed elsewhere. This method needs to examine only the PDG. We suspect that some other update problems can be solved in a similar manner. Most update problems may best be attacked with the methods of Ryder [25] and Wegman [34].

We assume throughout this paper that data flow information (if not the entire PDG) is retained in all libraries referenced by a module under development so that interprocedural flow analysis will not require reanalysis of existing modules.

6. Program Complexity Metrics

A software development environment should aid in the construction of understandable, maintainable code. As Curtis points out [7], the measurement of the psychological complexity of software is of increasing interest because of the rising proportion of overall system costs attributable to software. Measurements of software under development can provide important information to the programmer and perhaps permit some automatic restructuring. We feel that a metric based on the PDG and/or slicing would provide a step forward in this area and could be an effective tool in an integrated environment.

Much of the previous work in this area has been based on counts of some physical attributes of the source program. Measurements such as McCabe's cyclomatic metric [19] and Schneidewind and Hoffman's reachability metric [26] are based on graph theory considerations and measure characteristics of the control flow graph of the program. Halstead's [15] metrics are based on counts of operators and operands. Although reported research [4,5,6,14,17] has shown correlations between these metrics and attributes of the program that are considered related to program understandability, this type of metric can be easily criticized for oversimplification. Borrowing from linguistics terminology, we might say that these metrics consider surface characteristics of the program. We are, however, more interested in characteristics related to the "deep structure" of a program. The argument is that it is not just the number of operators and operands, the number or depth of conditionals, measure of the control flow graph, etc. that affect the clarity of a program. The nature of information flow must be considered.

Some complexity metrics have attempted to better account for the information flow in a program. An example is the information flow metric defined by Henry [17] which is based on the number of data values flowing into a procedure combined with the number of data values flowing out of the procedure. Although not described as a complexity metric, Dunsmore and Gannon had previously used a measure based on the number of variables that might potentially be accessed at a particular statement [10].

Weiser [35] has suggested several metrics based on slicing. These include: (1) coverage, a measure of the length of slices _versus_ the length of the program; (2) overlap, a measure of the number of statements in a slice which belong to no other slice; (3) clustering, the degree to which slices are reflected in the original code layout; (4) parallelism, the number of slices with few statements in common; and (5) tightness, the number of statements in every slice. A PDG, augmented with line numbers, can be used to compute these metrics efficiently.

The experiments performed by Weiser [35,36], as well as work by Soloway [11,28], has shown that programmers tend to group statements in ways based on other than sequential relationships when attempting to understand programs. In general, the criteria used for the groupings is related to data and control flow. Since this information is explicit in the PDG, this class of complexity metric may be most easily and accurately measured with the PDG.

Much of the difficulty in understanding a program, it could be hypothesized, is due to the difference between the groupings of statements that a programmer uses to understand the program (along with the effort required to obtain that grouping) and the groupings of statements that the sequential source program listing or programming environment presents. If one hypothesizes that programmers do use slices when debugging [36], then Weiser's third metric, clustering, appears to be of prime interest. In particular, a measure of the complexity of the slice along with its relationship to the original source code has appeal. Of course, many questions need to be answered to be able to define this metric. These include defining a measure of the complexity of a slice, describing the reflection of the slice in the code and combining the slice complexities to form an overall complexity metric for the program.

The feedback given to a programmer may help in the manual production of code which is easier to understand. It may even be possible to automatically transform the original code to reduce its psychological complexity by using a complexity metric and the PDG.

7. Practical Considerations

The amount of space required to represent a PDG is the dominating practical consideration for this work. Unfortunately, we cannot give very good estimates on anticipated space consumption since the size of the graph is related to the dependence structure, rather than to any easily measured surface feature of a program. We expect that the PDG for a given program will consume approximately three times as much space as a corresponding representation which represents each basic block as a DAG and in addition retains for each block data flow bit vectors for reaching definitions and live variables.

8. Conclusions

We have sketched some of the issues surrounding the use of the program dependence graph in a programming environment. In addition to supporting editing and translation tasks, a prime benefit is ease in generating accurate slices. Slices are obtained in linear time and are more accurate than those obtained with existing methods both because I/O is accounted for correctly and because irrelevant statements on multi-statement lines are not displayed. In addition, the form is advantageous because it can be interpreted or optimized, it is amenable to incremental data flow analysis, can be used to detect data flow anomalies when debugging and supports new software complexity metrics. Future work should include implementation of the ideas presented here as well as the exploration of other uses of the representation in a software development environment.

Acknowledgements

We thank Rick Longworth for reading a draft of this paper. His use of a program developed in [23] to construct limited slices inspired this paper. Jeanne Ferrante's comments on one section and Mark Wegman's pointing us to the LISPEDIT papers are appreciated.

REFERENCES

1. Alberga, C. N.; Brown, A. L.; Leeman, G. B., Jr.; Mikelsons, M. and Wegman, M. N. A program development tool. IBM J. of Res. & Dev. 28, 1 (Jan. 1984).

2. Allen, John Randal. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. Thesis, Rice University, Houston (April 1983) 181 pages.

3. Allen, J.R.; Kennedy, Ken; Porterfield, Carrie; and Warren, Joe. Conversion of control dependence to data dependence. Conf. Rec. 10th Ann. ACM Symp. on Princ. of Prog. Lang. Austin, Texas (Jan. 1983) 177-189.

4. Baker, Albert L. and Zweben, Stuart H. The use of software science in evaluating modularity concepts. IEEE TSE SE-5, 2 (March 1979) 110-120.

5. Basili, Victor R. and Phillips, Tsai-Yun. Evaluating and comparing software metrics in the software engineering laboratory. Proc. 1981 ACM Workshop/Symp. on Meas. and Eval. of Software Quality published in ACM SIGMETRICS Performance Evaluation Review 10, 1 (Spring 1981) 95-106.

6. Curtis, Bill; Sheppard, S. B.; and Milliman, P. Third time charm: stronger prediction of programmer performance by software complexity metrics. Proc. 4th IEEE Int. Conf. Soft. Eng. (1979) 356-360.

7. Curtis, Bill. Measurement and experimentation in software engineering. Proc. of the IEEE 68, 9 (Sept. 1980) 1144-1157.

8. Dennis, Jack B. First version of a data flow procedure language, revised Comp. Struc. Group Memo 93 (MAC Tech. Memo 61), Lab. for CS, MIT (May 1975) 21 pages.

9. Dennis, Jack B. Data flow supercomputers. IEEE Computer 13,11 (Nov. 1980) 48-56.

10. Dunsmore, H. E. and Gannon, J. D. Data referencing: an empirical investigation. IEEE Computer 12, 12 (Dec. 1979) 50-59.

11. Ehrlich, Kate and Soloway, Elliot. An empirical investigation of the tacit plan knowledge in programming. Research Report 236, Dept. of Computer Science, Yale Univ. (April 1982).

12. Ferrante, Jeanne and Ottenstein, Karl J. A program form based on data dependency in predicate regions. Conf. Rec. Tenth ACM Symp. on Princ. of Prog. Lang., Austin, Texas (Jan. 24-26, 1983) 217-236.

13. Ferrante, Jeanne; Ottenstein, Karl J.; and Warren, Joe D. The program dependence graph and its use in optimization. IBM Research Report RC-10208 (August 1983) 10 pages, revision to appear in the Proc. Sixth International Symp. on Programming, Toulouse, France (April 1984), Springer-Verlag. Detailed paper in preparation.

14. Feuer, Alan R. and Fowlkes, Edward B. Some results from an empirical study of computer software. Proc. of the 4th Int. Conf. on Soft. Eng., Munich, W. Germany, Sept. 17-19, 1979, pp. 351-355.

15. Halstead, M. H. Elements of Software Science, Elsevier North Holland, New York (1977).

16. Hennessy, J. L. Symbolic debugging of optimized code. ACM TOPLAS 5, 3 (July 1982) 323-344.

17. Henry, Sallie and Kafura, Dennis. Software structure metrics based on information flow. IEEE TSE SE-7, 5 (Sept. 1981) 510-518.

18. Howden, William E. Contemporary software development environments. CACM 25, 5 (May 1982) 318-329.

19. McCabe, T. J. A complexity measure. IEEE TSE SE-2 (1976) 308-320.

20. Mikelsons, M. Prettyprinting in an interactive environment. Proc. ACM SIGPLAN/SIGOA Symp. on Text Manipulation, Portland, OR (June 1981) 108-116, published as ACM SIGPLAN Notices.

21. Ottenstein, Karl J. and Ottenstein, Linda M. High-level debugging assistance via optimizing compiler technology (extended abstract), Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on High-Level Debugging (see [29] for full entry) 152-154.

22. Ottenstein, Karl J. An intermediate form based on a cyclic data-dependency graph. CS-TR 81-1, Math and Computer Sciences, Mich. Tech. Univ. (October 1981) 91 pages, under revision.

23. Ottenstein, Karl J. Data-flow graphs as an intermediate program form. Ph.D. Thesis, Computer Sciences, Purdue Univ. (August 1978) 283 pages.

24 Padua, David A.; Kuck, David J. and Lawrie, Duncan H. High-speed multiprocessors and compilation techniques. IEEE Trans. Comp. TC-29, 9 (Sept. 1980) 763-776.

25. Ryder, Barbara G. Incremental data flow analysis. Conf. Rec. Tenth ACM Symp. on Princ. of Prog. Lang., Austin, Texas (Jan. 24-26, 1983) 167-176.

26. Schneidewind, N. F. and Hoffman, H. M. An experiment in software error data collection and analysis. IEEE TSE SE-5, 3 (May 1979) 276-286.

27. Skedzielewski, Stephen and Glauert, J.R.W. IF1: An intermediate form for applicative languages, draft 7, November 21, 1983, Lawrence Livermore National Laboratory.

28. Soloway, Elliot; Ehrlich, Kate; Bonar, Jeffrey; and Greenspan, Judith. What do novices know about programming? Research Report 218, Dept. of Computer Science, Yale Univ. (Jan. 1982).

29. Tischler, Ron; Schaufler, Robin; Payne, Charlotte. Static analysis of programs as an aid to debugging. Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on High-Level Debugging, Pacific Grove, CA, March 20-23, 1983 published as ACM Software Engineering Notes 8, 4 (August 1983) and as ACM SIGPLAN Notices 18, 8 (August 1983) 155-158.

30. Treleaven, Philip C.; Hopkins, Richard P. and Rautenbach, Paul W. Combining data flow and control flow computing. The Computer Journal 25, 2 (1982) 208-217.

31. Warren, Joe D. A hierarchical basis for reordering transformations. Conf. Rec. 11th ACM Symp. Princ. Prog. Lang., Salt Lake City, Utah (January 1984).

32. Waters, Richard C. The programmer's apprentice: knowledge based program editing. IEEE Trans. Soft. Eng. SE-8, 1 (Jan. 1982) 1-12.

33. Waters, Richard C. Automatic analysis of the logical structure of programs. MIT AI-Lab TR-492, available as NTIS AD-A084 818 (December 1978) 207 pages.

34. Wegman, Mark. Summarizing graphs by regular expressions. Conf. Rec. Tenth ACM Symp. on Princ. of Prog. Lang., Austin, Texas (Jan. 24-26, 1983) 203-216.

35. Weiser, Mark. Program slicing. Proc. 5th Int. Conf. on Soft. Eng., San Diego, Calif., IEEE Computer Soc. Press (March 9-12, 1981) 439-449.

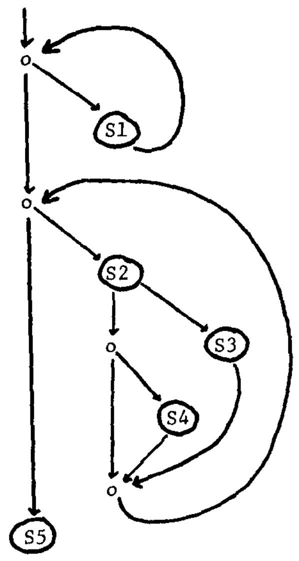
36. Weiser, Mark. Programmers use slices when debugging. CACM 25, 27 (July 1982) 446-452.

37. Zellweger, Polle T. An interactive high-level debugger for control-flow optimized programs, Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on High-Level Debugging (see [29] for full entry) 159-171.

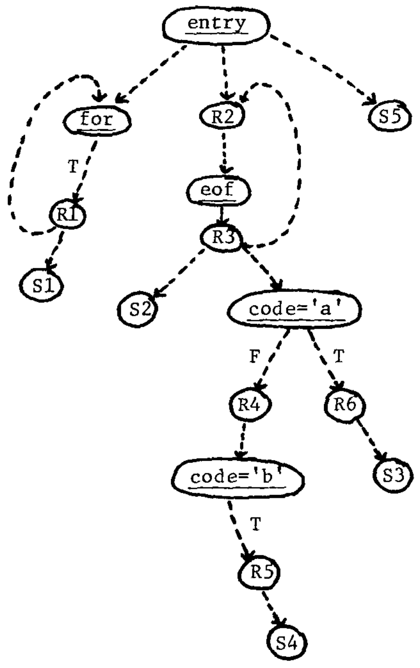
```
const
   last = 10;
...
begin
   for i := 1 to last do
      begin
         a[i] := 0;
         b[i] := 0
      end;

   while not eof do
      begin
         read (code, value);
         read (dummy);
         if code = 'a' then
            a[value] := a[value] + 1
         else if code = 'b' then
            b[value] := b[value] - 1
      end;
   writeln (a[last], b[last])
end.
```

Figure 1 - A Program Fragment



(a) Control Flow Graph



(b) PDG Control Subgraph

Figure 2 - Control Flow Representations

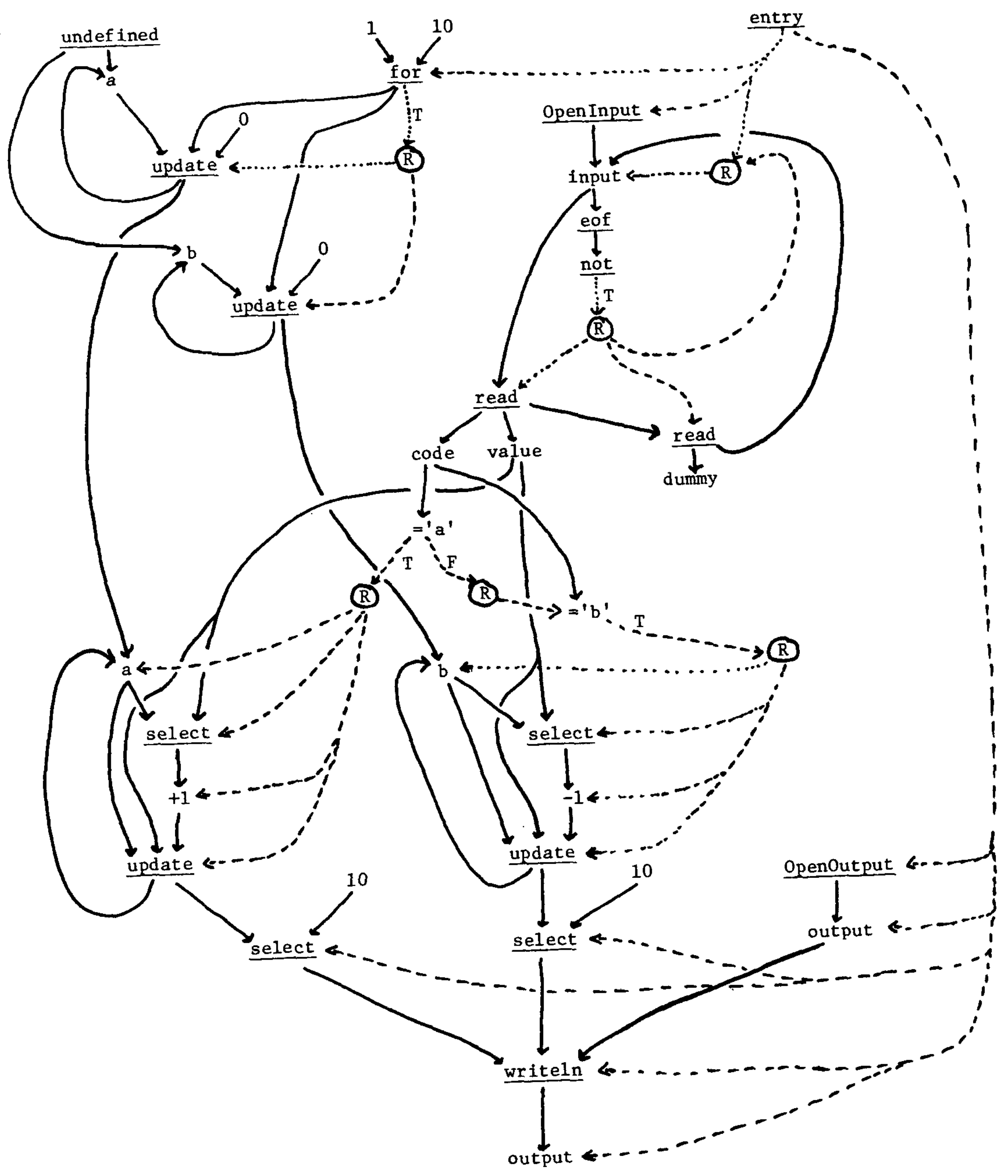


Figure 3 - PDG for Program in Figure 1