

A New Model of Program Dependences for Reverse Engineering

*Daniel Jackson and Eugene J. Rollins
School of Computer Science
Carnegie Mellon University*

Abstract

A dependence model for reverse engineering should treat procedures in a modular fashion and should be fine-grained, distinguishing dependences that are due to different variables. The program dependence graph (PDG) satisfies neither of these criteria. We present a new form of dependence graph that satisfies both, while retaining the advantages of the PDG: it is easy to construct and allows program slicing to be implemented as a simple graph traversal. We define ‘chopping’, a generalization of slicing that can express most of its variants, and show that, using our dependence graph, it produces more accurate results than algorithms based directly on the PDG.

Keywords

Reverse engineering, modularity, specifications, program slicing, dataflow dependence, program dependence graph.

1 Introduction

Many analyses and transformations of programs are based on dependence relationships, often represented by the program dependence graph (PDG). Originally devised for compilers, its novelty was to combine dataflow and control dependences in a single graph, making code optimizations easier to perform [FOW87]. More recently, the PDG has been adopted in soft-

ware engineering, for analyses whose outputs are intended not for compiler backends but for software developers. Program slicing in particular benefits greatly, being reduced to a simple reachability problem [OO84] far simpler than its original formulation [Wei84].

But the PDG is too coarse for software engineering applications. The notion of program dependences arose from the question of whether a compiler can reorder statements without affecting execution behaviour: a statement that reads a variable, for example, cannot be swapped with an earlier statement that writes it. For this purpose, dependences between variables are auxiliary, and, in the construction of the PDG, are discarded, leaving only dependences between statements. Questions about dependences between variables are thus inexpressible; one cannot determine from the PDG, for example, which variables at the start of a procedure might affect a given variable at the end.

Slicing suffers a loss of precision from the coarseness of the PDG. In Weiser’s formulation, the slice criterion identifies one or more variables at a given line, and the slice is a subprogram whose statements might affect the value of those variables just prior to execution of that line. In PDG formulations of slicing, such as [RY89], on the other hand, a slice criterion is a node of the graph—that is, a program statement—and the slice contains statements that might affect the value of any variable used by that statement. For the statement

$$a := b + c$$

a PDG-based slicer equates three distinct criteria: the definition of a , the use of b and the use of c .

This loss of precision can be overcome by exposing the internal structure of PDG nodes, reconstructing the dependences between variables from the syntax of the program statements. Two approaches are possible. One can adapt the slicing algorithm to accommodate special cases; a slice on the use of a variable, for example, can be translated into a slice on all the nodes containing definitions that reach that use. The structure of the PDG is untarnished, but the algorithm is no longer a simple graph traversal. Alternatively, one can adapt the PDG itself, splitting nodes to distinguish variables where necessary. To allow slicing on the final value of a variable, for instance, one can provide a ‘final use’ node for every variable

Address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Phone: (412) 268-5143. Fax: (412) 268-5576. Email: daniel.jackson@cs.cmu.edu. Research sponsored by a Research Initiation Award from the National Science Foundation (CCR-9308726), by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (F33615-93-1-1330). Composed on an Apple Macintosh in Microsoft Word and Aldus Intellidraw, and set in Quark Xpress. Text face generously provided by Bitstream, Inc.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGSOFT '94- 12/94 New Orleans LA USA
© 1994 ACM 0-89791-691-3/94/0012..\$3.50

in the program [RY89]. And to distinguish the uses and definitions of a procedure call, which is essential for reasonable interprocedural slices, one can insert a mock assignment node for every passing of a parameter, and for every reading and writing of a global by the called procedure [HRB90].

Program slicing is not alone; there are many analyses based on dependences for which the PDG is not ideal. We were unable to use PDGs, for example, in a tool that used unsatisfied dependences to detect bugs [Jac93] or in a differencing tool that compared programs according to their dependences [JL94]. Reverse engineering in particular demands two properties of a dependence model, neither of which the PDG satisfies:

1. *Procedures should have a modular representation.*

Queries and reports about procedure calls can then be implemented without complex mappings between the user's view of the code and the underlying dependence graph, and without compromising the coherence of the graph or the algorithms operating upon it. Procedures without code, such as library routines, can be smoothly accommodated (with dependence specifications to replace their bodies) and, in the absence of recursion, procedures can be analyzed independently.

2. *The dependences should be fine-grained,* relating not whole statements but, rather, definitions and uses of individual variables. Otherwise, queries and reports can only be cast in terms of program statements, and data structures are relegated to a secondary role.

The model should also retain the desirable properties of the PDG: simplicity, ease of construction and support for slicing as graph traversal.

To address these properties, our model differs from the PDG in two respects. First, the dependences relate instances of variables (definitions and uses at particular sites) rather than program statements. Second, a procedure call is modelled as a single node with summary dependences relating the variables it defines to the variables it uses. All statements get the same treatment, so a simple assignment has summary dependences too, even though these may be determined locally.

To demonstrate the utility of this model, we use it to define a generalization of slicing called ‘chopping’. Most slicing notions, such as backward and forward slicing, are easily expressed as special cases. Chopping produces smaller slices than simple PDG-based algorithms, without the complications of more precise algorithms that use the PDG but compensate in an ad hoc fashion for its weaknesses.

Throughout the paper, the construction and analysis steps are given in terms of relational expressions. The operators we use to compose and project relations are listed in an appendix. We prefer this formulation to the traditional one—giving explicit worklist algorithms—primarily because the resulting definitions are shorter, and perhaps simpler and more direct. Being specifications of a sort, they also admit, unlike algorithms, a variety of implementations. They are also good for prototyping. So long as the relational operators are implemented carefully, the analysis can be coded in a few lines by transcribing the definitions directly. Hardwiring the dependence relations speeds up the graph traversals, and, with suitable optimizations, enables the efficiency of the PDG to be matched.

2 A New Dependence Model

Figure 1 shows a program consisting of three procedures that access five global variables, a , b , r , f and i . The procedure *golden* sets r to the golden ratio by generating a fibonacci sequence (with calls to *step*) and calculating the ratio of each term b to its predecessor a (with calls to *check*). When r converges, *check* sets the flag f to false and the loop terminates with i equal to the number of iterations.

The diagram shows *golden*'s dependence graph. The boxes represent statements (such as the assignment $i := 0$) and conditional tests (*while f*); there are also special boxes, *entry* and *exit*, that model the calling context of the procedure.

Each box has ports labelled with variables, one for each variable used by the statement (at the top of the box) and one for each variable defined (at the bottom of the box). Since a procedure is called in a context that defines variables before the call and use them after, the entry box has a definition port for every variable, and the exit, a use port. There are three special symbols that also label ports: τ , a temporary that holds the result of a conditional test, γ , which represents a constant, and ϵ , which stands for ‘execution’.

Edges between ports denote dependences. An edge between boxes that connects program variables is a dataflow dependence. The edge between the i definition port of $i := 0$ and the i use port of *step*(), for instance, says that the use of i by *step*() may depend directly on a preceding definition by $i := 0$.

An edge between boxes that connects a τ port to an ϵ port is a control dependence. The edge connecting *while f* to *step*, for instance, indicates that the execution of *step* may depend on the result of evaluating the loop condition.

An edge inside a statement's box (shown dotted) represents a dependence brought about by the statement itself. In *step*(), for instance, the edge from the use of a to the definition of b says that an execution of *step* may use a to define b . There is no edge from the use of a to its definition, since a is always set to b and thus cannot depend on its previous value. Every variable that is defined in a statement box depends on the use of the special symbol ϵ . The edges from ϵ to a and b say that the definitions of a and b depend on whether or not *step*() is executed. When a statement defines a variable without reference to a use, a dependence on the constant symbol γ is introduced. The assignment $i := 0$, for instance, has an edge from γ to i since i is defined but the resulting value does not depend on the previous state.

A box's internal edges may be viewed as a dependence specification of the corresponding statement. For a primitive statement, the specification follows simply from the syntax of the statement, according to the semantics of the programming language. For a procedure call, the specification is a summary of the dependence graph of the procedure body. The dependence of b on a in *step*(), for instance, summarizes the chain in the body of *step* that passes back from the definition of b through the temporary t to the initial use of a . Procedures without code, such as library routines, can be given surrogate specifications in place of code to be incorporated directly [Jac93].

To see how the dependence edges fit together, let's slice on the use of a by the statement *step*(). That is, we want to determine which statements might affect the value of a just prior to

```

a, b: int := 1
r: real := 0
i: int
f: bool

```

```

proc step
  t: int
  t := a + b
  a := b
  b := t
  i := i + 1

```

```

proc check
  if r = b/a then
    f := false
  else
    r := b/a
  end

```

```

proc golden
  i := 0
  f := true
  while f do
    step()
    check()
  end

```

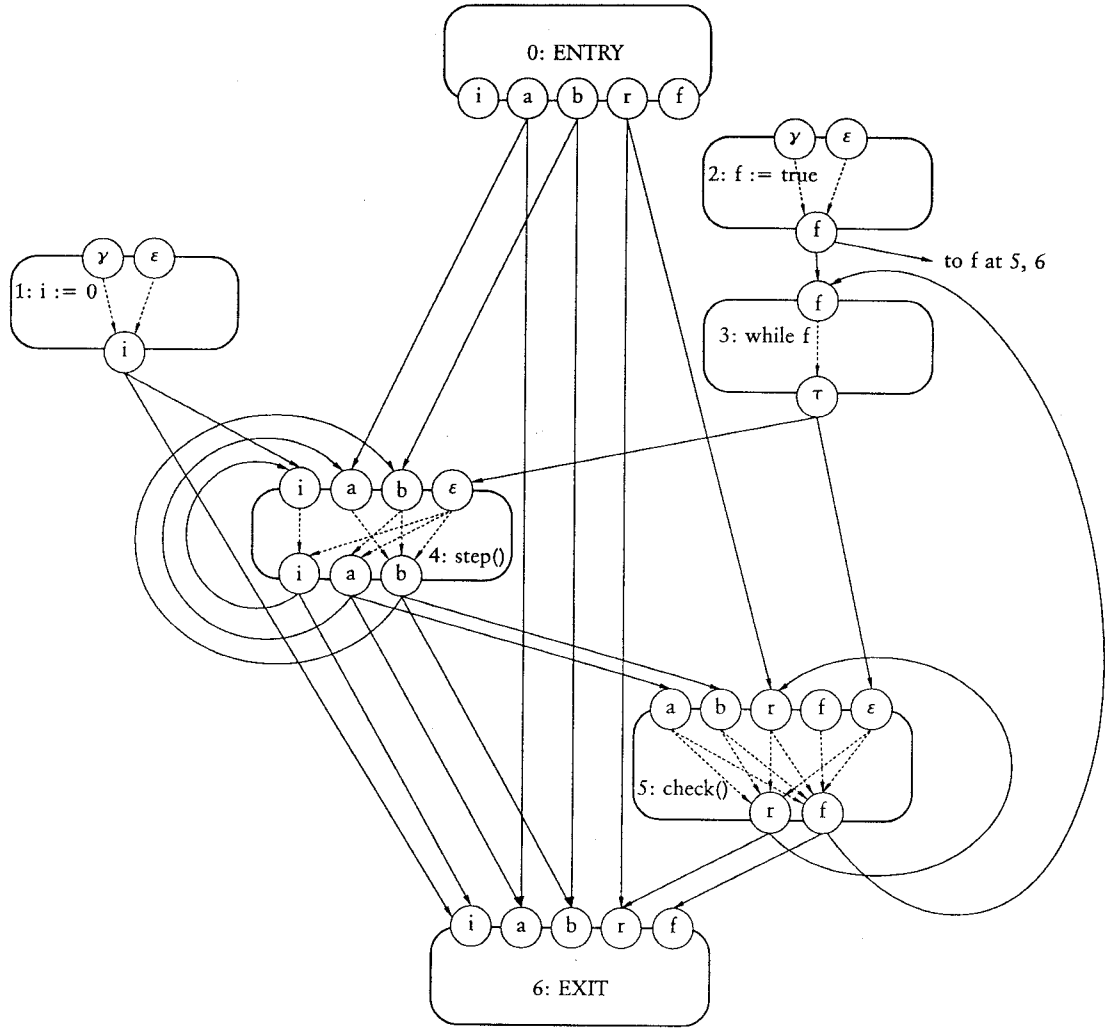


Figure 1: A program and the dependence graph of one procedure, *golden*

the call to *step*. Slicing, as in the PDG, is just graph traversal: we follow the edges backwards, marking boxes encountered on the way. The use of *a* has two incoming edges. One takes us to the definition of *a* at the entry. The other edge is a loop-carried dependence; it takes us to the definition of *a* by *step()* itself. Crossing the *step()* box to the relevant uses leads us to *b* and ϵ . Following the use of *b* takes us to the definition of *b* by the entry, and, via another loop-carried dependence, to the definition of *b* by *step()*. Crossing *step()* a second time brings in the uses of *a*, *b* and ϵ . We have already taken the paths from *a* and *b*, so we follow the edge incident on ϵ . This takes us to the loop condition, and subsequently to *f* := *true* and the call to *check*. The resulting slice includes every box except the exit and the assignment *i* := 0.

To see why variable uses and definitions must be distinguished, consider a slice at *step()* on the use of *i* instead of *a*. This should include *i* := 0. A PDG-based algorithm cannot generally make such distinctions without ad hoc additions, such as special nodes for the final use and initial definition of variables [RY89], mock assignments for passing parameters

and globals in and out of procedure calls [HRB90], and the splitting of nodes (or adaptation of the search algorithm) for separating the uses of a primitive statement. Our graph supports simple traversal algorithms without such tricks.

3 Formalization of the Graph

A dependence relates a variable at one program point to a variable at another. We could define a point before and after each statement, but it is simpler to define a single *site* for each statement, and separate uses and definitions by formalizing internal and external edges as two distinct relations. An *instance* is a pair consisting of a variable and a site; variables include the special symbols, and sites include the entry and exit:

$$\begin{aligned}
\text{Var} &= \text{ProgramVariable} \cup \{\gamma, \epsilon, \tau\} \\
\text{Site} &= \text{ProgramStmt} \cup \{\text{entry}, \text{exit}\} \\
\text{Instance} &= \text{Var} \times \text{Site}
\end{aligned}$$

It is convenient to split the external edges into two relations, a

dataflow dependence relation ud and a control dependence relation cd . A third relation du models the internal edges.

ud, cd, du : *Instance* $\leftrightarrow$ *Instance*

The ud relation connects a use of a variable at one site to a definition of the same variable at another; cd connects an execution of one statement to a temporary defined by another; and du connects a definition of a variable at one site to a use of another at the same site:

$$\begin{aligned} ud &\subseteq \{((x, i), (x, j)) \mid x \in \text{ProgramVariable} \wedge i, j \in \text{Site}\} \\ cd &\subseteq \{((\epsilon, i), (\tau, j)) \mid i, j \in \text{ProgramStmt}\} \\ du &\subseteq \{((x, i), (y, i)) \mid x, y \in \text{ProgramVariable} \wedge i \in \text{Site}\} \end{aligned}$$

All variables are defined at the entry and used at the exit:

$$\begin{aligned} \text{ProgramVariable} \times \{\text{entry}\} &\subseteq \text{dom } du \\ \text{ProgramVariable} \times \{\text{exit}\} &\subseteq \text{ran } du \end{aligned}$$

Sometimes we shall not want to distinguish control and dataflow dependences, so we give a name to their union:

$$ucd = ud \cup cd$$

Here are some examples. Let's identify sites by numbering the statements (as in Figure 1). The edge from the definition of a at the entry to its use at $\text{step}()$ belongs to ud , and is given by the pair

$$((a, 4), (a, 0)).$$

The internal edge from this use of a to the definition of b belongs to du , and is given by

$$((b, 4), (a, 4)).$$

The indirect dependence of the definition of b at $\text{step}()$ on its definition at the entry

$$((b, 4), (a, 0))$$

belongs to the composition $du \circ ud$. Similarly the definition of r at $\text{check}()$ depends indirectly on the use of f by the loop test because of the path

$$\begin{aligned} ((r, 5), (\epsilon, 5)) &\in du, \\ ((\epsilon, 5), (\tau, 3)) &\in cd, \\ ((\tau, 3), (f, 3)) &\in du \end{aligned}$$

summarized by the pair

$$((r, 5), (f, 3)) \in du \circ cd \circ du.$$

All indirect dependences belong to one of four closures. UU contains dependences of uses on uses; DD relates definitions to definitions; UD relates uses to definitions; and DU relates definitions to uses:

$$\begin{aligned} UU &= (ucd \circ du)^* \\ DD &= (du \circ ucd)^* \\ UD &= ucd \circ (du \circ ucd)^* \\ DU &= du \circ (ucd \circ du)^* \end{aligned}$$

The operator $*$ is the reflexive (and transitive) closure, so it includes direct dependences also, and in the case of UU and DD , a dependence of each use or definition on itself. (Both UU and DD actually contain dependences of every instance on itself, so $\text{dom } UU$, for instance, is not equal to $\text{dom } ud$, the set

of all uses.)

The naming convention allows a simple kind of type checking. Note how adjacent u 's and d 's are matched in the closure expressions. An expression like

$$ucd \circ ucd$$

makes no sense because it confuses uses and definitions; the second ucd treats as a use the definition in the range of the first.

The du relation, incidentally, should not be confused with du -chaining [ASU88]. We chose to order the relations so that membership of a pair $((x, i), (y, j))$ can be read ' x at i depends on y at j '. An element of a du -chain is a dependence of a use on a definition, and would thus belong to ud and not du .

Our treatment of modifications is also unconventional. Dependences on y allow the du relation to express, additionally, the set of variables modified by a statement, which would otherwise have to be maintained separately. These dummy dependences model not only assignments of constants, but any modification that has no explicit dependences (usually due to non-determinism, such as a read operation in which the input stream has no name).

4 Chopping: A Modular Generalization of Slicing

Chopping is a generalization of slicing. Although expressible as a combination of intersections and unions of forward and backward slices, chopping seems to be a fairly natural notion in its own right.

Two sets of instances form the criterion: *source*, a set of definitions, and *sink*, a set of uses. Chopping a program identifies a subset of its statements that account for all influences of *source* on *sink*. A conventional backward slice is a chop in which all the *sink* instances belong to the same site, and the *source* set contains every variable at every site.

A chop is confined to a single procedure. The instances in *source* and *sink* must be within the procedure, and chopping only identifies statements in the text of the procedure itself. We believe that, for reverse engineering at least, analyses should be modular, respecting the structure of the program. Since programmers tend to approach a new program one procedure at a time, it seems that a reverse engineering tool should do the same. The form of our dependence graph supports modular chopping quite naturally; by presenting not only relevant statements but relevant dependence edges too, the role of a procedure call in a chop can be explained to the user without requiring a foray into its body. Moreover, should a programmer want a traditional interprocedural slice that extends both into the code of called and calling procedures, the chop contains sufficient information (namely the relevant instances) to initiate further chops in different scopes.

Suppose, for example, we want to slice *golden* (Figure 1) on the use of a by the call to step . The criterion is

$$\begin{aligned} \text{source} &= \text{Var} \times \text{Site} \\ \text{sink} &= \{(a, 4)\} \end{aligned}$$

and the resulting chop is shown in Figure 2. By examining the edges in the graph, we can see why, for example, $\text{check}()$ is

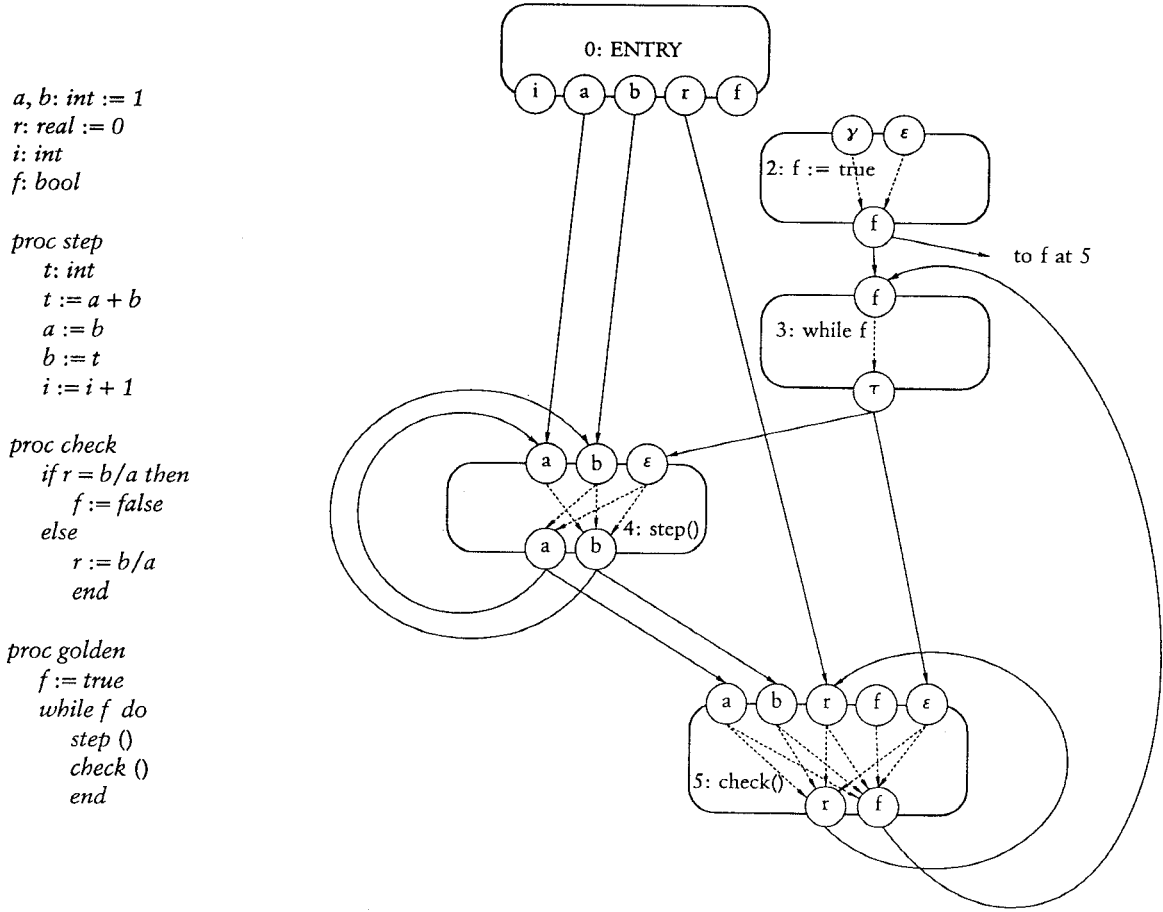


Figure 2: Slice of procedure *golden* (Figure 1) on use of *a* by *step()*

included: it defines *f*, the flag that controls the loop. To follow the slice into *check()*, we note that *check()*'s definitions of both *f* and *r* were included in the chop (since the definition of *f* depends on the use of *r*, via *du*, and the use of *r* depends on its definition in a previous loop iteration, via *ud*). The relevant statements inside *check* are then found by chopping it with

$source = Var \times Site$
 $sink = \{(f, exit), (r, exit)\}$

where *exit* now refers to the graph for *check*.

5 Formalization of Chopping

Rather than give explicit algorithms for the traversal of the dependence graph, we shall use the closures from Section 3 (and relational operators defined in the appendix). Recall that *UU* relates each use of a variable to all the other uses in the procedure it depends on, directly or indirectly. The projection

$UU(sink)$

thus defines the uses that might affect *sink*. Similarly, the definitions affected by *source* are found by projection under the inverse of *DD*:

$DD^{\sim}(source)$

The chop is obtained by selecting edges from the graph that connect relevant definitions to relevant uses:

$ud' = UU(sink) \triangleleft ud \triangleright DD^{\sim}(source)$
 $cd' = UU(sink) \triangleleft cd \triangleright DD^{\sim}(source)$
 $du' = DD^{\sim}(source) \triangleleft du \triangleright UU(sink)$

It can be presented in various ways. A tool might display the entire subgraph, showing all three sets of edges. It might show only the edges between boxes (*ud'* and *cd'*) or perhaps just the dataflow edges (*ud'*). A textual display would highlight the set of relevant statements, given by

$sites(UU(sink)) \cap sites(DD^{\sim}(source))$

where

$sites(I) = \{s \mid (s, x) \in I\}.$

Most slicing notions that have been proposed can be expressed as forms of chopping:

(1) a traditional backward slice on the use of variables V at site i :

$$\text{source} = \text{Var} \times \text{Site}, \text{sink} = V \times \{i\}$$

(2) a forward slice [YL88] on the definition of variables V at site i :

$$\text{source} = V \times \{i\}, \text{sink} = \text{Var} \times \text{Site}$$

(3) a backward slice on all uses and definitions of variables at site i [RY89]:

$$\text{source} = \text{Var} \times \text{Site}, \text{sink} = \text{Var} \times \{i\}$$

(4) and a backward slice on the final value or any definition of a variable v [GL91]:

$$\text{source} = \text{Var} \times \text{Site}, \text{sink} = \text{du}(\{v\} \times \text{Site}) \cup \{(v, \text{exit})\}.$$

The original formulation of slicing identifies the statements that affect the value of a variable just prior to the execution of a given statement [Wei84]. Since the variable need not be used by the statement, this criterion cannot be expressed in our model. The value of a variable at an arbitrary site is only well defined given a strong semantic interpretation of the program that requires that the statements be executed in their syntactic order. Many compiler optimizations exploit a weaker semantics in which independent statements can be reordered. It is no surprise that program dependences, originally devised precisely to justify such reorderings, cannot support this form of slicing.

6 Constructing the Dependence Graph

Each site in the dependence graph has a specification that summarizes the dependences between the variables it defines and the variables it uses. These dependences are the internal edges in Figure 1, and, labelled by site and collected over the graph, the du relation of Section 5.

Let the specification $\text{spec}(i)$ of a site i be a relation on variables

$$\text{spec}: \text{Site} \rightarrow (\text{Var} \leftrightarrow \text{Var})$$

where

$$(x, y) \in \text{spec}(i)$$

when x depends on y at site i , that is, the statement at site i uses y to define x . Given these specifications, the dependence graph is easily constructed. Each site has a set of uses

$$\text{uses}(i) = \text{dom spec}(i)$$

and a set of definitions

$$\text{defs}(i) = \text{ran spec}(i).$$

From these two sets, the dataflow and control dependence edges are computed by the same technique used to construct the PDG. A set of reaching definitions is found for each site: a definition of variable x at node i reaches a node j if there is a path from i to j with no intervening definition of x [ASU88].

For each of these, if x is used at j , the edge $((x, j), (x, i))$ is inserted into ud . The control dependence edges are a little trickier. A post-dominator tree is calculated that associates the site of each conditional with the set of sites whose execution it influences directly [FOW87]. For each edge in the tree from site i to site j , the edge $((\epsilon, j), (\tau, i))$ is inserted into cd .

The du relation is trivial to construct from the specifications; it presents the same information in a different way, and adds a dependence of each defined variable on the special symbol ϵ :

$$(x, y) \in \text{spec}(i) \Leftrightarrow ((x, i), (y, i)) \in \text{du} \wedge ((x, i), (\epsilon, i)) \in \text{du}$$

Now the novel part: constructing the specifications. A primitive statement has a specification given by its syntax in a manner determined by the programming language. The assignment

$$a := b + c$$

at site i , for instance, gives

$$\text{spec}(i) = ((a, b), (a, c)).$$

Library or system calls require specifications to be provided; a call that writes a string s to the screen

$$\text{put}(s)$$

for instance, might have

$$\text{spec}(i) = \{(screen, screen), (screen, s)\}.$$

For a call to a procedure whose code is available, the specification is obtained from the dependence graph of the called procedure. To do this, we identify, within the procedure body, uses that are reached by, and definitions that reach, statements in the calling context. Call these the *exposed* uses and definitions. We then calculate the dependences of exposed definitions on exposed uses, by abstracting away intermediate edges. To find the exposed instances, there is no need actually to consider dependences that cross the procedure call boundary, since prior definitions are modelled by the called procedure's entry site, and subsequent uses by its exit.

Suppose, for example, that we want to determine the specification of the call to *check* in *golden* (Figure 1). The dependence graph for *check* is shown in Figure 3. The exposed definitions are $(r, 3)$ and $(f, 2)$, and the exposed uses $(a, 1)$, $(b, 1)$, $(r, 1)$ and $(a, 3)$ and $(b, 3)$. The indirect dependences of $(f, 2)$ on $(a, 1)$, $(b, 1)$ and $(r, 1)$ are abstracted into the specification dependences

$$(f, a), (f, b) \text{ and } (f, r)$$

and the direct dependences of $(r, 3)$ on $(a, 1)$, $(b, 1)$, $(r, 1)$, $(a, 3)$ and $(b, 3)$ into

$$(r, a), (r, b) \text{ and } (r, r)$$

Two complications may arise in this calculation. An exposed definition (x, i) might have no corresponding exposed use, because it depends on the constant symbol y . In this case the specification must include

$$(x, y).$$

More subtly, if a variable's definition is conditional, the specifi-

```

proc check
  if r = b/a then
    f := false
  else
    r := b/a
  end

```

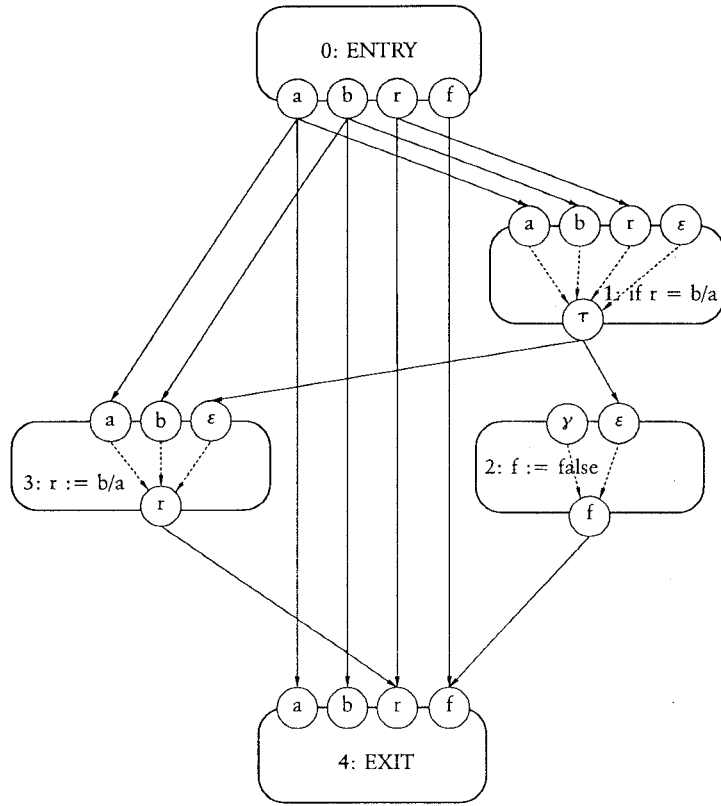


Figure 3: Procedure *check* and its dependence graph.

cation must include a dependence of the variable on itself. In *check*, for example, both definitions (f , 2) and (r , 3) are conditional—otherwise the graph would show no edges labelled f and r between entry and exit—adding

(f , f)

to the specification, (r , r) already having been included.

It might strike the reader as odd to handle conditional definitions in this way, rather than perhaps introducing some notion of a conditional dependence pair. If a definition of f were to immediately precede the call to *check*, the dependence graph of the caller would show no edges from that definition beyond the call, and yet, if the body of *check* were inlined, there would be a definition-free path and that definition would reach subsequent uses. Although strange, this follows directly from abstracting the procedure call. The statements

```
if b1 then b2 := true
```

and

```
b2 := b1 or b2
```

behave identically, and as bodies of a procedure, would have the same specification

```
{(b2, b2), (b2, b1)}
```

even though, when inlined, only the second would kill preceding definitions.

To formalize the construction of the specification, we start by noting that the definitions at the entry and the uses at the exit are not effects of the procedure body, but model the calling context. So we define

$$\begin{aligned} \text{entryDefs} &= \text{Var} \times \{\text{entry}\} \\ \text{exitUses} &= \text{Var} \times \{\text{exit}\}. \end{aligned}$$

The exposed definitions are those that reach the exit from the body of the procedure

$$\text{expDefs} = \text{ud}(\text{exitUses}) \setminus \text{entryDefs}$$

and the exposes uses are those within the body reached by the entry

$$\text{expUses} = \text{ud}^{\sim}(\text{entryDefs}) \setminus \text{exitUses}$$

To find how the definitions depend on the uses, we restrict the *DU* closure to dependences of exposed definitions on exposed uses

$$\text{DU}' = \text{expDefs} \triangleleft \text{DU} \triangleright \text{expUses}$$

and project out the sites, obtaining a relation between variables:

$$\text{deps} = \{(x, y) \mid \exists i, j \in \text{Site}. ((x, i), (y, j)) \in \text{DU}'\}$$

If a variable used at the exit has a direct dependence on a definition of the same variable at the entry, it may be invariant:

$$\text{invs} = \{x \in \text{Var} \mid ((x, \text{exit}), (x, \text{entry})) \in \text{ud}\}$$

On the other hand, a variable that has an exposed definition might be modified

$$mods = \{x \in Var \mid \exists i \in Site. (x, i) \in expDefs\}.$$

The final specification contains pairs for three kinds of definition:

$$\begin{aligned} spec = & deps \\ & \cup (mods \setminus (invs \cup dom\ deps)) \times \{y\} \\ & \cup \{(x, x) \mid x \in mods \cap invs\} \end{aligned}$$

The first term gives the definitions that depend on exposed uses; the second gives definitions that have no explicit dependence; and the third adds self-dependences for variables that are potentially unmodified.

Some details have been omitted in this explanation. Variables local to the scope of the called procedure (such as t in *step* in Figure 1) are simply dropped from the specification. Call-by-value parameters are handled by renaming formals to actuals when the *du* relation is constructed from the specification.

Recursion prevents the simple bottom-up construction of specifications, but the scheme is easily extended. Initially, the specification of each procedure is calculated from its dependence graph on the assumption that recursively called procedures have empty specifications. The specifications are then recalculated, using the new approximations of called procedures, repeatedly until a fixpoint is reached.

This calculation can be efficiently implemented as a graph traversal. The specification of a procedure is found by tracing backwards from the exit, maintaining at each instance in the graph the set of exit variables it influences. When a variable is already in the set, it is propagated no further; each variable is thus propagated at most once along each edge. For k variables and n instances, this bounds the execution time by kn^2 . Most statements only access a few variables, so n varies linearly with the size of the program. Experiments will determine how efficient this approach is, but we suspect it to be no worse than the best existing methods [R+94].

7 The Chopshop Tool

The dependence model was designed for *Chopshop*, a tool we have built to analyze C programs. Chopshop is written in ML and runs as subprocess of *emacs* 19. Chopping criteria are given by clicking on variables that appear in the text buffer and selecting options from a menu.

Chops are presented in two ways: by highlighting of relevant instances in the code, and by display of a dependence graph (with the help of *dot*, a utility that generates postscript from an adjacency list representation, and *ghostview*, a postscript previewer). Sites are represented as simple nodes, and *ud* edges alone are shown, each with a label to show which variable carries the dependence.

Graphs of even the smallest chops tend to be huge, but we have found that a few simple abstraction mechanisms—such as eliding primitive statements and folding calls of the same procedure—reduce the size drastically without adversely affecting the graph’s utility [JR94].

Our current challenge is to incorporate aliasing in a modular fashion, probably combining ideas taken from abstract interpretation schemes [LH88] and more specialized techniques [PLR94].

8 Related Work

A number of dependence representations have been developed that treat procedures in a modular fashion. The value dependence graph (VDG) [W+94] associates summary edges with calls and supports fine-grained slicing [Ern94]. Being designed primarily as an intermediate representation for a compiler, the model does not seem to incorporate codeless procedures as smoothly. It is also functional, loops being replaced by recursive calls and mutation of variables being modelled explicitly with store update operations. More significantly, the VDG is focused on operations where our graph is focused on variables. A slicer based on the VDG highlights operator symbols and procedure names rather than variables, and cannot identify which variables are responsible for dependences between calls.

The dependence specifications originate in our previous work, where they were used to find bugs in a procedure by comparing its calculated and expected dependences [Jac91]. Dependence relations were also used in the Spade tool to detect dataflow anomalies and to extract ‘partial statements’ similar to program slices [BC85]. These relations have a different form, however; lacking a distinction between uses and definitions of the same variable occurrence, they seem unable to handle procedure calls with side effects.

Moriconi and Winkler’s inference rule system for determining the scope of a program change [MW90] defines a dependence relation too, but implicitly as a set of inference rules over the syntax. Procedure calls are abstracted, since the proof of a dependence due to a call can be built from rules applied to its body. It might be interesting to see how this logic is related to our dependence graph.

Finally, Wilde and Huitt’s ‘external dependency graphs’ seem to be identical to our specifications. They are mentioned briefly in [WH91], along with a variety of other dependence relations, but with no explanation of how they are constructed and used.

Acknowledgments

Discussions with Bill Griswold of UCSD and Mike Ernst of Microsoft Research occasioned a complete rewriting of this paper. Bill Griswold pressed us to concede that PDG-based slicing can be elaborated to match the precision of chopping, but clarified for us, based on his experience building a PDG-based tool, the practical problems this causes. Mike Ernst explained his VDG-based slicer to us, and pointed out the optimization that allows recursive calls to be analyzed efficiently. The referees provided unusually helpful comments; one, in particular, found an error in Section 8, confirming our plan eventually to justify our model theoretically, following the examples of [CF89] and [PC90].

Appendix: Relational Operators

The paper uses the Z syntax [Spi89] for operators on sets and relations. Throughout, s and t are sets of elements of type T , and p and q are binary relations on T .

set difference	$s \setminus t = \{e: s \mid e \notin t\}$
identity relation	$I = \{(t, t) \mid t: T\}$
domain	$\text{dom } p = \{a: T \mid \exists b: T. (a, b) \in p\}$
range	$\text{ran } p = \{b: T \mid \exists a: T. (a, b) \in p\}$
inverse	$p^\sim = \{(b, a) \mid (a, b) \in p\}$
composition	$p \circ q = \{(a, b) \mid \exists z: T. (a, z) \in p \wedge (z, b) \in q\}$
domain restriction	$s \triangleleft p = \{(a, b) \in p \mid a \in s\}$
range restriction	$p \triangleright s = \{(a, b) \in p \mid b \in s\}$
image	$p(s) = \{b \mid \exists a: s. (a, b) \in p\} = \text{ran } (s \triangleleft p)$

The reflexive and transitive closure of p

$$p^* = I \cup p \cup (p \circ p) \cup (p \circ p \circ p) \cup \dots$$

is the smallest relation containing p that is reflexive ($I \subseteq p^*$) and transitive ($p^* \circ p^* \subseteq p^*$).

References

- [ASU88] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, March 1988.
- [BC85] Jean-Francois Bergeretti and Bernard A. Carre. Information-flow and dataflow analysis of while-programs. *ACM Trans. on Programming Languages and Systems*, 7(1), January 1985, pp. 37–61.
- [CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. *Proc. ACM Symposium on Programming Language Design and Implementation*, 1989.
- [Ern94] Michael D. Ernst. *Practical fine-grained static slicing of optimized code*. Technical report MSR-TR-94-14, Microsoft Research, Redmond, Wa., July 1994.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3), July 1987, pp. 319–349.
- [GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. on Software Engineering*, 17(8), August 1991, pp. 751–761.
- [HRB90] Susan Horwitz, Thomas Reps and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1), January 1990, pp. 26–60.
- [Jac91] Daniel Jackson. Aspect: An Economical Bug Detector. *Proc. International Conf. Software Engineering*, Austin, Texas, May 1991, pp. 13–22.
- [Jac93] Daniel Jackson. Abstract Analysis with Aspect. *Proc. International Symposium on Software Testing and Analysis*, Cambridge, Mass., June 1993, pp. 19–27.
- [JL94] Daniel Jackson and David A. Ladd. Semantic diff: a tool for summarizing the effects of modifications. *Proc. International Conf. on Software Maintenance*, Vancouver, October 1994.
- [JR94] Daniel Jackson and Eugene J. Rollins. Abstraction mechanisms for pictorial slicing. *Proc. of Workshop on Program Comprehension*, Washington, D.C., November 1994.
- [LH88] James J. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *Proc. ACM Conf. on Principles of Programming Language Design and Implementation*, June 1988.
- [MW90] Mark Moriconi and Timothy C. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Trans. on Software Engineering*, 16(9), September 1990, pp. 980–992.
- [OO84] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984. *ACM SIGPLAN Notices* 19(5), pp. 177–184, May 1984.
- [PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE Trans. on Software Engineering*, 16(9), September 1990, pp. 965–979.
- [PLR94] Hemant D. Pande, William A. Landi and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. on Software Engineering*, 20(5), May 1994, pp. 385–403.
- [R+94] Thomas Reps, Susan Horwitz, Mooly Sagiv and Genevieve Rosay. *Speeding up slicing*. Technical report D-214, Datalogisk Institut, University of Copenhagen, 1994.
- [RY89] Thomas Reps and Wu Yang. The semantics of program slicing and program integration. *Proc. Colloquium on Current Issues in Programming Languages*, Barcelona, March 1989. *Lecture Notes in Computer Science* 352, pp. 360–374, Springer-Verlag, New York.
- [Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., 1989.
- [W+94] Daniel Weise, Roger F. Crew, Michael Ernst and Bjarne Steensgaard. *Value dependence graphs: representation without taxation*. Technical report MSR-TR-94-03, Microsoft Research, Redmond, Wa., April 1994.
- [Wei84] Mark Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4), July 1984, pp. 352–357.
- [WH91] Norman Wilde and Ross Huitt. A reusable toolset for software dependency analysis. *Journal of Systems and Software*, Vol. 14, 1991, pp. 97–102.
- [YL88] S. Yau and S.S. Liu. *Some approaches to logical ripple-effect analysis*. Software Engineering Research Center, SERC-TR-24F, University of Florida, October 1988.