Information-Flow and Data-Flow Analysis of **while**-Programs

JEAN-FRANCOIS BERGERETTI and BERNARD A. CARRÉ
University of Southampton

Until recently, information-flow analysis has been used primarily to verify that information transmission between program variables cannot violate security requirements. Here, the notion of information flow is explored as an aid to program development and validation.

Information-flow relations are presented for **while**-programs, which identify those program statements whose execution may cause information to be transmitted from or to particular input, internal, or output values. It is shown with examples how these flow relations can be helpful in writing, testing, and updating programs; they also usefully extend the class of errors which can be detected automatically in the "static analysis" of a program.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.4 [**Software Engineering**]: Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments

General Terms: Information flow, data flow, analysis of programs, automatic error detection, debugging, software maintenance

1. INTRODUCTION

As was noted by the authors of Euclid [19], "... most programs presented to verifiers are actually wrong; considerable time can be wasted looking for proofs of incorrect programs before discovering that debugging is still needed. This problem can be reduced (although not eliminated) by judicious testing, which is generally the most efficient way to demonstrate the presence of bugs ...."

The testing process can be assisted, for instance, by run-time checks of assertions [19], but in general it remains a rather haphazard way of revealing the errors in a program. For this reason, there is currently much interest in extending program flow analysis methods, such as data-flow analysis, to enable these methods to detect a larger class of errors than they can uncover at the present time [18].

This paper describes some information-flow relations that are easily constructed for a **while**-program and that are helpful both in program testing and in checking the consistency of assertions with a program text. They also usefully

Authors' address: Department of Electronics and Information Engineering, University of Southamptom, Southampton SO9 5NH, England.

```
⟨program⟩ ::= ⟨compound statement⟩
⟨statement⟩ ::= ⟨empty statement⟩ | ⟨assignment statement⟩ | ⟨compound statement⟩ |
                ⟨conditional statement⟩ | ⟨repetitive statement⟩
⟨empty statement⟩ ::=
⟨assignment statement⟩ ::= ⟨variable⟩ := ⟨expression⟩
⟨compound statement⟩ ::= begin ⟨statement⟩ {; ⟨statement⟩} end
⟨conditional statement⟩ ::= if ⟨expression⟩ then ⟨statement⟩ else ⟨statement⟩ |
                if ⟨expression⟩ then ⟨statement⟩
⟨repetitive statement⟩ ::= while ⟨expression⟩ do ⟨statement⟩
```

Fig. 1. Syntax of Pascal-like programming language.

extend the class of errors which can be detected automatically in the static analysis of a program.

In Section 2, the information-flow relations are presented, with their construction rules for assignment, compound, conditional, and repetitive statements; by repeated application of these rules it is possible to obtain the flow relations for the complete statement part of a procedure or program. For any program statement S we have three binary information-flow relations. One of these, which associates values of program variables on entry to S with values of variables on exit from S, is well known in the context of computer system security [4, 6, 7]. However, we also construct (a) a relation between the values of the program variables on entry to S and the values of the ⟨expression⟩ parts within S, and (b) a relation between the values of ⟨expression⟩ parts within S and the values of the program variables on exit from S. These relations enable us to identify the program statements which may play a role in the transmission of information, either from or to particular program variables.

Section 3 describes several ways in which these relations can help one to "comprehend" a program text. For instance, in informally checking an assertion involving some variable v at a particular point in a program, or in trying to understand why in a test run the value of v is erroneous at a particular point, we can use the information-flow relations to extract automatically a "partial program" composed of those statements which may affect the value of v at the point of interest.

Section 4 describes applications of the flow relations in the static analysis of programs for the automatic detection of errors and anomalies. It then draws a comparison between this form of analysis and global data-flow analysis, explaining the significance of some of the flow relations in terms of "reaching definitions" and "upward-exposed uses" of program variables [10, 12, 15]. It will be seen that, although the information yielded by the two methods is somewhat similar in nature, the flow relations reveal a larger class of "ineffective statements" than the class of anomalies (i.e., unused definitions) detected by the usual form of data-flow analysis.

The final section explains how the flow methods can be extended, for instance, to programs with procedure and function calls.

The techniques described use only syntactic information (including a specification of which program variables are referenced and defined by each statement). They therefore suffer from the weakness inherent in all syntactic methods—they

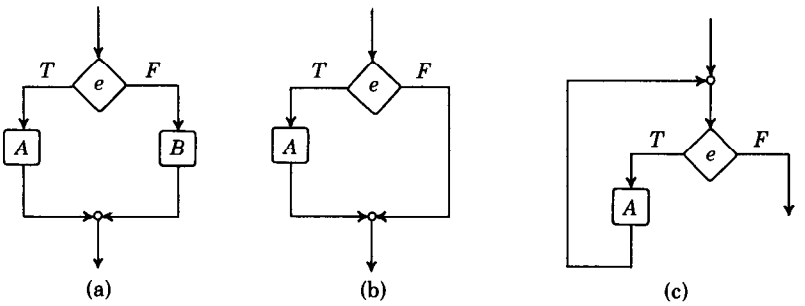


Fig. 2. Control-flow graphs of program statements. (a) **if** e **then** A **else** B. (b) **if** e **then** A. (c) **while** e **do** A.

are "conservative," in that they may indicate possible data dependencies even where the program semantics preclude these. Nevertheless, as the examples demonstrate, the flow relations provide a useful aid to program analysis.

2. INFORMATION-FLOW RELATIONS

For ease of exposition we first present our methods of information- and data-flow analysis with reference to the simple Pascal-like programming language whose syntax is given in Figure 1. We need not specify the grammar or semantics of expressions, but it is assumed that the evaluation of an expression has no side effects.

The control flow in any program statement S in this language can be represented by a *control-flow graph* $G(S)$, with one node corresponding to each assignment statement in S, one node corresponding to the ⟨expression⟩ part of each conditional and repetitive statement in S, and arcs representing the possible transfers of control (see Figure 2).

We denote by V the set of all variables in a program, and we let E denote the set of all instances of an ⟨expression⟩ in the program. Also, for each expression $e \in E$, we let $\Gamma(e)$ stand for the set of all variables which appear in e.

2.1 Defined and Preserved Variables

We say that (the execution of) a statement S *may define* a variable v if S contains any assignments to v, and we denote the set of variables which S may define by D_S. Also, we say that a statement S *may preserve* (the value of) a variable v if in $G(S)$ there exists a path from the entry to the exit which is definition-clear for v (i.e., which does not traverse any assignments to v). We denote the set of variables which S may preserve by P_S.

The sets D_S and P_S can be constructed immediately for an empty statement (these are defined by formulas (T1) and (T2) of Table I[1]) and for an assignment statement (see (T6) and (T7)). If S is a sequence of statements $(A; B; \ldots)$, then the sets D_S and P_S are easily constructed from the sets $D_A, D_B, \ldots$ and $P_A, P_B, \ldots$ associated with the constituents of S (see (T11) and (T12)). The same is true

[1] Table I contains formulas (T1)–(T30).

Table I. Definitions of the Information-Flow Relations

Empty Statements. For an empty (or "skip") statement S,

$D_S = \phi$	(T1)	$P_S = V$	(T2)
$\lambda_S = \phi$	(T3)	$\mu_S = \phi$	(T4)
$\rho_S = \iota$	(T5)		

Assignment Statements. For an assignment statement S, which assigns a value to v and whose expression part is e,

$D_S = \{v\}$	(T6)	$P_S = V - \{v\}$	(T7)
$\lambda_S = \Gamma(e) \times \{e\}$	(T8)	$\mu_S = \{(e, v)\}$	(T9)
$\rho_S = (\Gamma(e) \times \{v\}) \cup (\iota - \{(v, v)\})$	(T10)		

Sequences of Statements. For a sequence S of two statements $(A; B)$,

$D_S = D_A \cup D_B$	(T11)	$P_S = P_A \cap P_B$	(T12)
$\lambda_S = \lambda_A \cup \rho_A \lambda_B$	(T13)	$\mu_S = \mu_A \rho_B \cup \mu_B$	(T14)
$\rho_S = \rho_A \rho_B$	(T15)		

Conditional Statements. For a statement S of the form: **if** e **then** A **else** B

$D_S = D_A \cup D_B$	(T16)	$P_S = P_A \cup P_B$	(T17)
$\lambda_S = (\Gamma(e) \times \{e\}) \cup \lambda_A \cup \lambda_B$	(T18)	$\mu_S = (\{e\} \times (D_A \cup D_B)) \cup \mu_A \cup \mu_B$	(T19)
$\rho_S = (\Gamma(e) \times (D_A \cup D_B)) \cup \rho_A \cup \rho_B$	(T20)		

For a statement S of the form: **if** e **then** A

$D_S = D_A$	(T21)	$P_S = V$	(T22)
$\lambda_S = (\Gamma(e) \times \{e\}) \cup \lambda_A$	(T23)	$\mu_S = (\{e\} \times D_A) \cup \mu_A$	(T24)
$\rho_S = (\Gamma(e) \times D_A) \cup \rho_A \cup \iota$	(T25)		

Repetitive Statements. For a statement S of the form: **while** e **do** A

$D_S = D_A$	(T26)	$P_S = V$	(T27)
$\lambda_S = \rho_A^*((\Gamma(e) \times \{e\}) \cup \lambda_A)$	(T28)	$\mu_S = (\{e\} \times D_A) \cup \mu_A \rho_A^*((\Gamma(e) \times D_A) \cup \iota)$	(T29)
$\rho_S = \rho_A^*((\Gamma(e) \times D_A) \cup \iota)$	(T30)		

for conditional statements (see (T16), (T17) and (T21), (T22)) and repetitive statements (see (T26) and (T27)).

2.2 Relations between the Variables and Expressions of a Program

Our method of analyzing programs is based on the successive constructions of three binary relations for each statement S of a program in turn:

λ_S, from V to E
μ_S, from E to V
ρ_S, from V to V.

Before presenting their precise definitions, it will be helpful to give a general indication of the significance of each relation. First, for any variable $v \in V$ and any expression $e \in E$, the condition $v \lambda_S e$ can be (loosely) interpreted as "the value of v on entry to S *may be used* in the evaluation of the expression e in S." As a simple example, the statement

if $y > 0$ **then** $x := y + z$

contains two expressions ($y > 0$ and $y + z$). The value of y on entry to this statement *may be used* in evaluating both expressions; the entry value of z *may be used* in evaluating the second expression only. For the statement S consisting of two assignments

begin $x := y + z$; $w := 2 * x$ **end**

the values of y and z on entry to S *may be used* in evaluating the expression $y + z$; we also say that these values *may be used* in evaluating the second expression $2 * x$, since the value of the first expression determines the value of the variable x in the second. We do not say that the value of x on entry to S *may be used* in the evaluation of the second expression, since this entry value is "killed" by the first assignment statement.

As a final example, let us consider the compound statement

begin if $x > 0$ **then** $y := 1$; $z := 2 * y$ **end**

Here we say that the entry value of x *may be used* in the evaluation of the final expression $2 * y$, because depending on the entry value of x, an assignment may be made to y, and this assigned value would be used in the final evaluation. In this example we also say that the value of y on entry to S *may be used* in evaluating the expression $2 * y$, since this value will be used if no assignment is made to y.

Considering now our second relation μ_S, the condition $e \mu_S v$ signifies that "a value of the expression e in S *may be used* in obtaining the value of the variable v on exit from S." As an example, in the statement

if $w > 0$ **then** $x := y + z$

both the expressions $w > 0$ and $y + z$ *may be used* in obtaining the exit value of x: the value of the first expression determines whether an assignment is to be made to x, and if so, the second expression defines the assigned value.

Our final relation ρ_s can be expressed in terms of λ_S, μ_S, and P_S as follows:

$$\rho_S = \lambda_S \mu_S \cup \Pi_S, \qquad \text{where } \Pi_S = \{(v, v) \in V \times V \mid v \in P_S\}. \tag{2.1}$$

Thus the condition $v \rho_S v'$ (which may be read as "the value of v on entry to S *may be used* in obtaining the value of v' on exit from S") signifies that either

(i) the entry value of v *may be used* in obtaining the value of some expression e in S, which in turn *may be used* in obtaining the exit value of v', or
(ii) $v = v'$, and S *may preserve* v.

As an example, for the statement

if $x > 0$ **then** $y := z$

we have $x \rho_S y$ because the entry value of x *may be used* in evaluating the first expression ($x > 0$), and the value of this expression *may be used* in determining the exit value of y. We also have $x \rho_S x$ and $y \rho_S y$, because S *may preserve* x and y.

We now give our definitions of the relations λ_S, μ_S, and ρ_S for each type of program statement. (A summary of all these definitions appears in Table I.)

Empty Statements. Since an empty statement does not contain any expressions, $\lambda_S = \phi$ and $\mu_S = \phi$. Then, in accordance with (2.1) and (T2) we define ρ_S to be the equality relation on V, viz.

$$\iota = \{(v, w) \in V \times V \mid v = w\}.$$

Assignment Statements. We say that (the value of) a variable v' on entry to an assignment statement S *may be used* in (the evaluation of) the expression e of this statement if and only if v' appears in e; the set of ordered pairs (v', e) for which this condition is satisfied constitute the relation λ_S for the statement (cf. (T8)). Also, we say that (the value of) e *may be used* in obtaining (the value of) the assigned variable v on exit from S; this relationship between the expression and assigned variable is represented by the relation μ_S (cf. (T9)). The formula for ρ_S follows immediately from (2.1) and (T7), (T8), and (T9).

Sequences of Statements. By definition, the relation λ_S for a sequence of statements $(A; B)$ comprises every ordered pair (v, e) such that either

(i) e is in A and $v \lambda_A e$, or
(ii) e is in B, and there exists some variable v' such that $v \rho_A v'$ and $v' \lambda_B e$.

Similarly, by definition, μ_S consists of every ordered pair (e, v) such that either

(i) e is in A and there exists some variable v' such that $e \mu_A v'$ and $v' \rho_B v$, or
(ii) e is in B and $e \mu_B v$.

By substituting the formulas (T12), (T13), (T14) for P_S, λ_S, and μ_S in (2.1), we obtain

$$\begin{aligned}\rho_S &= (\lambda_A \cup \rho_A \lambda_B)(\mu_A \rho_B \cup \mu_B) \cup (\Pi_A \cap \Pi_B) \\ &= \lambda_A \mu_A \rho_B \cup \lambda_A \mu_B \cup \rho_A \lambda_B \mu_A \rho_B \cup \rho_A \lambda_B \mu_B \cup (\Pi_A \cap \Pi_B). \qquad (2.2)\end{aligned}$$

Now $\lambda_A \mu_B = \phi$, since no expression e can belong to both A and B; for the same reason, $\lambda_B \mu_A = \phi$; hence the term $\rho_A \lambda_B \mu_A \rho_B$ is null also. The term $\Pi_A \cap \Pi_B$ can be expressed as $\Pi_A \Pi_B$, since Π_A and Π_B are subsets of the equality relation ι. Consequently (2.2) can be written as

$$\begin{aligned}\rho_S &= \lambda_A \mu_A \rho_B \cup \rho_A \lambda_B \mu_B \cup \Pi_A \Pi_B \\ &= \lambda_A \mu_A (\lambda_B \mu_B \cup \Pi_B) \cup (\lambda_A \mu_A \cup \Pi_A) \lambda_B \mu_B \cup \Pi_A \Pi_B \\ &= \lambda_A \mu_A \lambda_B \mu_B \cup \lambda_A \mu_A \Pi_B \cup \lambda_A \mu_A \lambda_B \mu_B \cup \Pi_A \lambda_B \mu_B \cup \Pi_A \Pi_B \\ &= \lambda_A \mu_A \lambda_B \mu_B \cup \lambda_A \mu_A \Pi_B \cup \Pi_A \lambda_B \mu_B \cup \Pi_A \Pi_B \\ &= (\lambda_A \mu_A \cup \Pi_A)(\lambda_B \mu_B \cup \Pi_B) \\ &= \rho_A \rho_B.\end{aligned}$$

It is sufficient to give the rules for constructing λ_S and μ_S for a sequence of only two statements, since a longer sequence $(A; B; C; \ldots)$ may be considered to

be nested, for instance, in the form $(((A; B); C); \ldots)$. It is easily verified that the information-flow relations obtained for a sequence of statements $(A; B; C; \ldots)$ are independent of the order in which they are combined.

Conditional Statements. Let S be a conditional statement of the form

if e **then** A **else** B

Then by definition (the value of) a variable v on entry to S *may be used* in evaluating an expression e' (and we have $v \lambda_S e'$) if and only if

(i) e' is the Boolean expression e of the conditional statement and $v \in \Gamma(e)$, or
(ii) e' is in A and $v \lambda_A e'$, or
(iii) e' is in B and $v \lambda_B e'$.

Similarly, by definition (the value of) an expression e' *may be used* in obtaining the value of a variable v on exit from a conditional statement S (and we have $e' \lambda_S v$) if and only if

(i) e' is the Boolean expression e of S, and either A or B *may define* v, or
(ii) e' is in A and $e' \mu_A v$, or
(iii) e' is in B and $e' \mu_B v$.

These definitions are given in symbolic form in (T18) and (T19). From (2.1) and (T17), (T18), (T19),

$$\begin{aligned}\rho_S &= ((\Gamma(e) \times \{e\}) \cup \lambda_A \cup \lambda_B)((\{e\} \times (D_A \cup D_B)) \cup \mu_A \cup \mu_B) \cup (\Pi_A \cup \Pi_B) \\ &= (\Gamma(e) \times (D_A \cup D_B)) \cup \lambda_A \mu_A \cup \lambda_B \mu_B \cup \Pi_A \cup \Pi_B \\ &= (\Gamma(e) \times (D_A \cup D_B)) \cup \rho_A \cup \rho_B.\end{aligned}$$

A conditional statement S of the form

if e **then** A

may be regarded as a statement of the form **if** e **then** A **else** B in which B is an empty statement, for which the information-flow relations are given by (T1)–(T5). Using these, the formulas (T16)–(T20) simplify to (T21)–(T25).

Repetitive Statements. For a statement S of the form

while e **do** A

the formulas for λ_S, μ_S, and ρ_S are obtained by assuming that S is equivalent to an infinite sequence of nested conditional statements of the form

```
if e then
begin
  A;
  if e then
  begin
    A;
     .
     .
     .
  end
end
```

To derive the required formulas it will be convenient to use the notations

$$\chi_e = \Gamma(e) \times \{e\}, \qquad \psi_e = \{e\} \times D_A, \qquad \omega_e = \Gamma(e) \times D_A.$$

Now by repeated application of (T23) and (T13),

$$\begin{aligned}\lambda_S &= \chi_e \cup \lambda_A \cup \rho_A(\chi_e \cup \lambda_A \cup \rho_A(\chi_e \cup \lambda_A \cup \rho_A(\cdots))) \\ &= (\iota \cup \rho_A \cup \rho_A^2 \cup \rho_A^3 \cup \cdots)(\chi_e \cup \lambda_A) \\ &= \rho_A^*(\chi_e \cup \lambda_A) \\ &= \rho_A^*((\Gamma(e) \times \{e\}) \cup \lambda_A)\end{aligned}$$

where ρ_A^* is the transitive closure of ρ_A.

It is convenient next to derive ρ_S^*, from (T25) and (T15), whose repeated applications give

$$\begin{aligned}\rho_S &= \omega_e \cup \iota \cup \rho_A(\omega_e \cup \iota \cup \rho_A(\omega_e \cup \iota \cup \rho_A(\cdots))) \\ &= (\iota \cup \rho_A \cup \rho_A^2 \cup \rho_A^3 \cup \cdots)(\omega_e \cup \iota) \\ &= \rho_A^*((\Gamma(e) \times D_A) \cup \iota).\end{aligned}$$

Finally, to obtain μ_S, we make repeated use of (T24) and (T14), which give

$$\mu_S = (\psi_e \cup \mu_A \rho_s) \cup (\psi_e \cup \mu_A \rho_s) \cdots = \psi_e \cup \mu_A \rho_s$$

and therefore from (T30) we obtain

$$\mu_S = (\{e\} \times D_A) \cup \mu_A \rho_A^*((\Gamma(e) \times D_A) \cup \iota).$$

2.3 Construction of the Information-Flow Relations

Given the parse tree of a program generated by the grammar of Figure 1, it is easily possible to compute its information-flow relations by obtaining these for each statement of the program in turn in the course of a post-order traversal [1] of the tree. On visiting a tree node which represents an assignment statement, we obtain its information-flow relations directly from the formulas (T6)–(T10); on visiting a node which represents a compound, conditional, or repetitive statement S, we use the appropriate formulas of Table I to derive the information-flow relations for S from those of its constituent statements. (These are immediate successors of S in the parse tree, which will already have been visited in the post-order traversal.) In this way, at each stage it is only necessary to retain the information-flow relations for the immediate successors of the tree nodes on the path from the tree root to the node currently being visited.

As an example, Figure 3a is the extended Euclidean algorithm, in the form of a Pascal procedure, as presented by Jensen and Wirth [14]. The information-flow relations for the body of its **while**-statement are given in Figure 3b. (Here each relation is represented by its Boolean relation matrix, with empty rows and columns omitted.) The corresponding relations for the complete **while**-statement are given in Figure 3c, and those for the entire procedure are shown in Figure 3d.

```
Expression
Number
          procedure GCD(m, n: integer; var x, y, z: integer);
          var a1, a2, b1, b2, c, d, q, r,: integer; {m ≥ 0, n > 0}
          begin {Greatest Common Divisor x of m and n,
                 Extended Euclid's Algorithm}
  1–4       a1 := 0; a2 := 1; b1 := 1; b2 := 0;
  5, 6      c := m; d := n;
  7         while d ≠ 0 do
            begin {a1 * m + b1 * n = d, a2 * m + b2 * n = c,
              gcd(c, d) = gcd(m, n)}
  8, 9        q := c div d; r := c mod d;
 10, 11       a2 := a2 − q * a1; b2 := b2 − q * b1;
 12, 13       c := d; d := r;
 14–16        r := a1; a1 := a2; a2 := r;
 17–19        r := b1; b1 := b2; b2 := r
            end;
 20–22      x := c; y := a2; z := b2
            {x = gcd(m, n) = y * m + z * n}
          end
```

Fig. 3a. Extended Euclidean algorithm.

Another example is given in Figure 4, which shows the information-flow relations of a procedure for hardware integer division.

3. TOOLS FOR PROGRAM ANALYSIS

In this section we describe some ways in which the information-flow relations can help one to "comprehend" a program text. These aids are useful in the course of writing a program (for instance, in making a preliminary "informal" check that the text of a procedure is consistent with its pre- and post-assertions). They are useful also in debugging (where one must first "comprehend" a program feature which causes an error, before eliminating it), and in updating a program (where one must be aware of all the possible effects of any proposed program modification).

In the following, for any program or subprogram P we denote by V^{i} the set of its variables whose values are to be "imported" in some way. (For instance, in the terminology of Ada, P could be a procedure and V^{i} the set of its parameters of mode *in* or *in out*). Similarly, we denote by V^{o} the set of variables whose final values are to be "exported." (Thus, in Ada terms, P could be a procedure and V^{o} the set of variables whose values are assigned to actual parameters of mode *in out* or *out*, as a result of the execution of P. Alternatively, P could be a function subprogram, having no side effects, with V^{o} comprising the single variable whose value is returned.)

3.1 Partial Statements

Let v be any program variable, and for any statement S let E_S^v be the set of expressions

$$E_S^v = \{e \in E \mid e \,\mu_S\, v\}.$$

$D_A = \{a1, a2, b1, b2, c, d, q, r\}; \quad P_A = \{m, n, x, y, z\}$

$$\lambda_A = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{array} \\ \left[\begin{array}{ccccccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}\right] \end{array}$$

$$\mu_A = \begin{array}{c} \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \end{array} \begin{array}{c} \begin{array}{cccccccc} a1 & a2 & b1 & b2 & c & d & q & r \end{array} \\ \left[\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

$$\rho_A = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \\ m \\ n \\ q \\ r \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} a1 & a2 & b1 & b2 & c & d & m & n & q & r & x & y & z \end{array} \\ \left[\begin{array}{ccccccccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

Fig. 3b. Information-flow relations for the body of the **while**-statement of Figure 3a.

In words, E_S^v is the set of expressions in S whose values *may be used* in obtaining the value of v on exit from S.

Now let S^v be the statement derived from S by replacing every statement within S which does not contain some member of E_S^v by an empty statement. Clearly, the statement S^v is equivalent to S in the sense that, for any set of input

$D_S = \{a1, a2, b1, b2, c, d, q, r\}; \quad P_S = V$

$$\lambda_S = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{array} \\ \left[\begin{array}{ccccccccccccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right] \end{array}$$

$$\mu_S = \begin{array}{c} \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \end{array} \begin{array}{c} \begin{array}{cccccccc} a1 & a2 & b1 & b2 & c & d & q & r \end{array} \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

$$\rho_S = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \\ m \\ n \\ q \\ r \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} a1 & a2 & b1 & b2 & c & d & m & n & q & r & x & y & z \end{array} \\ \left[\begin{array}{ccccccccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

Fig. 3c. Information-flow relations for the **while**-statement of Figure 3b.

values, the values of v on exit from S and S^v are identical; we describe S^v as the *partial statement of* S *(associated with* v*)*.

As an example, for the statement S of the extended Euclidean algorithm in Figure 3, the partial statements S^x and S^y associated with the greatest common

$D_S = \{a1, a2, b1, b2, c, d, q, r, x, y, z\}$; $P_S = \{m, n, q, r\}$

λ_S

	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
m	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
n	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

μ_S

	$a1$	$a2$	$b1$	$b2$	c	d	q	r	x	y	z
1	1	1	0	0	0	0	0	0	0	1	0
2	1	1	0	0	0	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	0	0	1
4	0	0	1	1	0	0	0	1	0	0	1
5	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	0	0	1	1	0	1	1
9	1	1	1	1	1	1	1	1	1	1	1
10	1	1	0	0	0	0	0	0	0	1	0
11	0	0	1	1	0	0	0	0	0	0	1
12	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1
14	1	1	0	0	0	0	0	0	0	1	0
15	1	1	0	0	0	0	0	0	0	1	0
16	1	1	0	0	0	0	0	0	0	1	0
17	0	0	1	1	0	0	0	1	0	0	1
18	0	0	1	1	0	0	0	1	0	0	1
19	0	0	1	1	0	0	0	1	0	0	1
20	0	0	0	0	0	0	0	0	1	0	0
21	0	0	0	0	0	0	0	0	0	1	0
22	0	0	0	0	0	0	0	0	0	0	1

ρ_S

	$a1$	$a2$	$b1$	$b2$	c	d	m	n	q	r	x	y	z
m	1	1	1	1	1	1	1	0	1	1	1	1	1
n	1	1	1	1	1	1	0	1	1	1	1	1	1
q	0	0	0	0	0	0	0	0	1	0	0	0	0
r	0	0	0	0	0	0	0	0	0	1	0	0	0

Fig. 3d. Information-flow relations for the Euclidean algorithm of Figure 3a.

denominator (gcd) and the multiplier y are given in Figures 5a and 5b, respectively. Again, in the case of the integer-division algorithm of Figure 4, we see from the relation μ that all its statements may be used in computing the quotient q; the partial statement S^r associated with the remainder r is given in Figure 6. Of course, because our analysis method is purely syntactic, a partial statement S^v may contain statements which in reality cannot affect the exit value of v. Nevertheless, it will be obvious, even from these very small examples, that the extraction of partial statements can be useful as an aid to understanding how

```
Expression
Number
          procedure division(x, y: integer; var q, r: integer);
          var w: integer;
          begin {(x ≥ 0) ∧ (y > 0)}
1–3         r := x; q := 0; w := y;
            {∃ n(w = 2^n * y) ∧ (x = q * w + r) ∧ (0 ≤ r)}
4, 5        while w ≤ x do w := 2 * w;
            {∃ n(w = 2^n * y) ∧ (x = q * w + r) ∧ (0 ≤ r < w)}
6           while w ≠ y do
            begin
7, 8          q := 2 * q; w := w div 2;
9             if w ≤ r then
                begin
10, 11            r := r − w; q := 1 + q
                end
            end
            {(x = q * y + r) ∧ (0 ≤ r < y)}
          end
```

Fig. 4a. Hardware integer-division algorithm.

$$D_S = \{q, r, w\} \qquad P_S = \{x, y\}$$

$$\lambda_S = \begin{array}{c} \\ x \\ y \end{array} \begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$$\mu_S = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{ccc} q & r & w \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

$$\rho_S = \begin{array}{c} \\ x \\ y \end{array} \begin{array}{c} \begin{array}{ccccc} q & r & w & x & y \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Fig. 4b. Information-flow relations for the hardware integer-division algorithm of Figure 4a.

the exported values of particular program variables are obtained. Our notion of a partial statement appears to be very similar to that of a "program slice," which was presented informally by Werser [23].

The notion of a partial statement leads to a further useful notion, of "interdependence" between program variables: we say that two variables v and v' are *independent* in a program statement S if

$$E_S^v \cap E_S^{v'} = \phi.$$

Expression Number	
	begin
5, 6	$c := m$; $d := n$;
7	**while** $d \neq 0$ **do**
	begin
9	$r := c$ **mod** d;
12, 13	$c := d$; $d := r$
	end;
20	$x := c$
	end

(a)

Expression Number	
	begin
1, 2	$a1 := 0$; $a2 := 1$;
5, 6	$c := m$; $d := n$;
7	**while** $d \neq 0$ **do**
	begin
8, 9	$q := c$ **div** d; $r := c$ **mod** d;
10	$a2 := a2 - q * a1$;
12, 13	$c := d$; $d := r$;
14–16	$r := a1$; $a1 := a2$; $a2 := r$
	end;
21	$y := a2$
	end

(b)

Fig. 5. Partial statements of the extended Euclidean algorithm of Figure 3. (a) Partial statement S^x. (b) Partial statement S^y.

Fig. 6. Partial statement S^r of the hardware integer-division algorithm of Figure 4.

Number	
	begin
	$r := x$; $w := y$;
1, 3	**while** $w \leq x$ **do** $w := 2 * w$;
4, 5	**while** $w \neq y$ **do**
6	**begin**
	$w :=$ w **div** 2;
8	**if** $w \leq r$ **then** $r := r - w$
9, 10	**end**
	end

Variables that are not independent are said to be (*weakly*) *dependent* on each other; and a variable v is said to be *strongly dependent* on a variable v' if

$$E_S^v \supseteq E_S^{v'}.$$

As an example, in the integer-division algorithm of Figure 4, the quotient q is strongly dependent on the remainder r, and r is strongly dependent on w, but the converses of these statements do not apply.

Knowledge of the interdependence of exported values again facilitates comprehension of a program and is helpful in choosing testing strategies and in detecting errors. For instance, knowledge of the fact that in the integer-division algorithm, $E_S^q \supseteq E_S^r$, with $E_S^q - E_S^r = \{e_2, e_7, e_{11}\}$, would enable one rapidly to locate an error, if in testing the algorithm one obtained correct values for the remainder but incorrect values for the quotient.

3.2 Values of Variables within a Program

The technique of the previous section can be extended to assist in establishing how the value of a variable v is obtained at some point within a program. (This may be necessary, for instance, in checking a loop invariant involving v or in searching for an error which causes the value of v to be incorrect at a particular point.) In this situation, we temporarily introduce at the point of interest an

assignment $v' := v$ to a new variable v'; the relevant statements are then all contained in the partial statement associated with v'.

3.3 Information Propagation within a Program

In analyzing a program, for instance to update it, one often wishes to know all the possible ways in which the value of some variable v at a particular point within the program may affect the subsequent program execution. (This problem arises, for instance, if one envisages inserting or modifying a particular assignment to v.)

To solve this problem, we may temporarily introduce a new program variable v' and insert at the point of interest an assignment statement $v := v'$. Then, in the statement part S of the program, the set of all expressions whose evaluations may employ the value of v at x (assuming that all program paths are executable) is

$$\{e \in E \mid v' \lambda_S e\}$$

and the set of variables for which the computation of exit values may employ the value of v at x (assuming that all paths are executable) is

$$\{v \in V \mid v' \rho_S v\}.$$

3.4 Input–Output Relations

For any program or subprogram P, the ρ-relation indicates those members of V^i whose values on entry to P *may be used* in obtaining the value, on exit from P, of each member of V^o. Thus a tabulation of ρ provides a useful indication of whether the relationships between input and output values are as required. As will be explained below, the evaluation of ρ for a procedure P also makes it possible to perform the type of analysis described here for a program in which P is embedded.

4. AUTOMATIC DETECTION OF ERRORS AND ANOMALIES

In this section we first describe a number of tests that can be performed on the λ-, μ-, and ρ-relations of a program to reveal particular kinds of programming errors. We then relate these tests to the well-known techniques of data-flow analysis.

4.1 Ineffective Statements

Let us suppose that S is the statement part of a program or subprogram P, and let V^o be the set of variables whose final values are exported. Then if S contains any expression e such that $e \tilde{\mu}_S v$ for all $v \in V^o$, the statement associated with e could be replaced by an empty statement without affecting the values exported by P; we describe such statements as being *ineffective.*

As an illustration, in the extended Euclidean algorithm of Figure 3, for which $V^o = \{x, y, z\}$, all the statements are effective. However, if the statement $r := b1$ were erroneously written as $r := a1$, the relation μ_S for the algorithm would be

$\mu_S =$

	$a1$	$a2$	$b1$	$b2$	c	d	q	r	x	y	z
1	1	1	1	1	0	0	0	0	0	1	1
2	1	1	1	1	0	0	0	0	0	1	1
3	0	0	1	0	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0	1
5	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	0	0	1	1	0	1	1
9	1	1	1	1	1	1	1	1	1	1	1
10	1	1	1	1	0	0	0	1	0	1	1
11	0	0	1	0	0	0	0	0	0	0	0
12	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1
14	1	1	1	1	0	0	0	1	0	1	1
15	1	1	1	1	0	0	0	1	0	1	1
16	1	1	1	1	0	0	0	1	0	1	1
17	0	0	1	1	0	0	0	1	0	0	1
18	0	0	1	0	0	0	0	0	0	0	0
19	0	0	1	1	0	0	0	0	0	0	1
20	0	0	0	0	0	0	0	0	1	0	0
21	0	0	0	0	0	0	0	0	0	1	0
22	0	0	0	0	0	0	0	0	0	0	1

Fig. 7. μ-relation for modified Euclidean algorithm.

that shown in Figure 7, which indicates that the statements 3, 11, and 18 are ineffective.

4.2 Ineffective Imported Values

Let P be a subprogram with specified sets V^{i} and V^{o} of imported and exported variables, respectively. If for any variable $v \in V^{\mathrm{i}}$ we have $v \bar{\rho}_S v'$ for all $v' \in V^{\mathrm{o}}$, then the imported value of v cannot be used in the evaluation of any exported value; we describe such an imported value v as being *ineffective.*

4.3 Undefined Variables

Let S be the statement part of a program or subprogram P, and let V^{i} be the set of variables for which initial values are to be imported by P.

If, for any variable $v \notin V^{\mathrm{i}}$, we have $v \lambda_S e$ for some e in S, then an undefined value (of v) *may be used* in the evaluation of e. As an illustration, if in the extended Euclidean algorithm of Figure 3 we had accidentally written the second assignment as $a1 := 2$, this procedure would have the λ-relation, shown in Fig. 8a, which indicates the possible use of an undefined value of $a2$ in evaluating the expressions 10, 14–16, and 21.

As an alternative to finding those expressions whose evaluation may involve an undefined variable—directly or indirectly—it is possible to find those expressions which may refer *directly* to an undefined variable. For this purpose we introduce a relation θ_S from V to E, where $v \theta_S e$ if and only if $v \in \Gamma(e)$ and $G(S)$

	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$a2$	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	1	0
m	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
n	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a)

	5	6	10	21
$a2$	0	0	1	1
m	1	0	0	0
n	0	1	0	0

(b)

	5	6	10
$a2$	0	0	1
m	1	0	0
n	0	1	0

(c)

Fig. 8. Relations λ, θ, and $\tilde{\theta}$ for the modified Euclidean algorithm. (a) Relation λ. (b) Relation θ. (c) Relation $\tilde{\theta}$.

contains a path, from its entry point to the node associated with e, that is definition-clear for v. This relation can easily be computed at the same time as the relations λ_S, μ_S, and ρ_S by using the formulas given in Table II. To illustrate its use, the relation θ_S for the Euclidean algorithm of Figure 3, with its second assignment statement changed to $a1 := 2$, is shown in Figure 8b; this reveals possible direct references to undefined variables in the expressions 10 and 21.

Of course, in applying these tests it must be remembered that the paths of a program may not all be executable, and therefore neither of the conditions $v \lambda_S e$ or $v \theta_S e$ imply that in some program execution the entry value of v is necessarily used in evaluating e; whether this happens can only be established (if at all) by considering the program semantics. However, we consider the matter to be of sufficient importance to be brought to a programmer's attention.

Another useful relation is the relation $\tilde{\theta}$ from V to E, where $v \tilde{\theta}_S e$ if and only if $v \in \Gamma(e)$ and in $G(S)$, all paths from the entry point to the node associated with e are definition-clear for v. In programming terms, the conditions $v \tilde{\theta}_S e$ signifies that the evaluation of e *always* involves a direct reference to the entry value of v; thus if $v \notin V^i$, the evaluation of e always involves a reference to an undefined variable. The relation $\tilde{\theta}_S$ is easily calculated, using the formulas given in Table II. As an illustration, the relation $\tilde{\theta}_S$ for the Euclidean algorithm of Figure 3, with its second assignment statement changed to $a1 := 2$, is shown in Figure 8c; this reveals that the evaluation of expression 10 always involves the use of an undefined value of $a2$.

4.4 Analysis of Repetitive Statements

In computing the information-flow relations for a complete program, we successively obtain these relations for the body A of each repetitive statement **while** e **do** A. From the relations λ_A, μ_A, and ρ_A for the body of a repetitive statement S, it is easily possible to determine whether S contains certain kinds of errors and "anomalies" in the following way.

Let $R(A)$ be the graph of the relation ρ_A (that is, the graph with node set V and arc set ρ_A), and let $\tilde{R}(A)$ be the subgraph of $R(A)$ generated by D_A (i.e., the

Table II. Expressions for the Relations θ_S and $\tilde{\theta}_S$

Empty Statements. For an empty (or "skip") statement S,

$$\theta_S = \phi; \qquad \tilde{\theta}_S = \phi.$$

Assignment Statements. For an assignment statement S, which assigns a value to v and whose ⟨expression⟩ part is e,

$$\theta_S = \Gamma(e) \times \{e\}; \qquad \tilde{\theta}_S = \Gamma(e) \times \{e\}.$$

Sequences of Statements. For a sequence S of two statements $(A; B)$,

$$\theta_S = \theta_A \cup \{(v, e) \in \theta_B \mid v \in P_A\}; \qquad \tilde{\theta}_S = \tilde{\theta}_A \cup \{(v, e) \in \tilde{\theta}_B \mid v \notin D_A\}.$$

Conditional Statements. For a statement S of the form: **if** e **then** A **else** B

$$\theta_S = (\Gamma(e) \times \{e\}) \cup \theta_A \cup \theta_B; \qquad \tilde{\theta}_S = (\Gamma(e) \times \{e\}) \cup \tilde{\theta}_A \cup \tilde{\theta}_B.$$

For a statement S of the form: **if** e **then** A

$$\theta_S = (\Gamma(e) \times \{e\}) \cup \theta_A; \qquad \tilde{\theta}_s = (\Gamma(e) \times \{e\}) \cup \tilde{\theta}_A.$$

Repetitive Statements. For a statement S of the form: **while** e **do** A

$$\theta_S = (\Gamma(e) \times \{e\}) \cup \theta_A; \qquad \tilde{\theta}_S = \{(v, e') \in ((\Gamma(e) \times \{e\}) \cup \tilde{\theta}_A) \mid v \notin D_A\}.$$

graph with node set D_A and arc set $\rho_A \cap (D_A \times D_A)$). A variable $v \in D_A$ is said to be *stable* (*with respect to* S) if on $\tilde{R}(A)$ there is no path to v from any variable which lies on a cycle. The *stability index* $\alpha_S(v)$ of a variable v which is stable in S is defined as the length (i.e., the number of arcs) of a longest path on $\tilde{R}(A)$ which terminates on v.

As an example, in the repetitive statement of the algorithm given in Figure 9a, none of the variables are stable. However, if the statement $r := c$ **mod** d is omitted, the relation ρ_A for the loop body becomes that shown in Figure 9b, and the corresponding graph $\tilde{R}(A)$ is that of Figure 9c; from this graph we see that the variables d, c, and q are stable, with stability indices 0, 1, and 2, respectively. (Stable nodes, with their indices, can easily be determined using the algorithm of Figure 10, which is a modified form of the algorithm commonly employed for finding node rank in acyclic graphs [5].)

In programming terms, the notion of stability has the following significance. Let us suppose that for some set of input values to S, some number $h > 0$ of iterations are performed, and for each variable $v \in V$, let us denote by $v^{(k)}$ the value of v after the kth execution of A. Now the value of any variable v which does not belong to D_A cannot be modified in the execution of A. Alternatively, if $v \in D_A$, then A contains an assignment to v, but in this case if $\alpha_S(v) = 0$ then $v' \bar{\rho}_A v$ for all $v' \in D_A$, which implies that

$$v^{(k+1)} = v^{(k)}, \qquad \text{for } k = 1, 2, \ldots, h - 1.$$

If $v \notin D_A$ and $\alpha_S(v) = 1$, then for any variable v' such that $v' \rho_A v$, either $v' \in D_A$ or $\alpha(v') = 0$, which implies that

$$v^{(k+1)} = v^{(k)}, \qquad \text{for } k = 2, 3, \ldots, h - 1.$$

By extension of this argument we find that for any stable variable v,

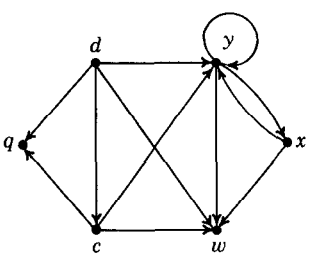
$$v^{(k+1)} = v^{(k)}, \qquad \text{for } k = \alpha_S(v) + 1, \quad \alpha_S(v) + 2, \ldots, h - 1.$$

```
Expression
Number
          function inverse(b, p: integer): integer;
          var c, d, q, r, w, x, y: integer;
          begin
 1, 2       c := p; d := b;
 3, 4       x := 0; y := 1;
 5          while d ≠ 1 do
            begin
 6, 7         q := c div d; r := c mod d;
 8            w := x − q * y;
 9, 10        c := d; d := r;
11, 12        x := y; y := w
            end;
13, 14      if y < 0 then y := y + p;
15          inverse := y
          end
```

(a)

	p	b	c	d	q	r	w	x	y
p	1	0	0	0	0	0	0	0	0
b	0	1	0	0	0	0	0	0	0
c	0	0	0	0	1	0	1	0	1
d	0	0	1	0	1	0	1	0	1
q	0	0	0	0	0	0	0	0	0
r	0	0	0	1	0	1	0	0	0
w	0	0	0	0	0	0	0	0	0
x	0	0	0	0	0	0	1	0	1
y	0	0	0	0	0	0	1	1	1

(b)



(c)

Fig. 9. (a) Algorithm for finding multiplicative inverse of b **mod** p. (b) ρ-Relation for the body of the **while**-statement in Figure 9a, with the statement $r := c$ **mod** d omitted. (c) Graph $\tilde{R}(A)$ for the modified **while**-statement.

As an example, for the repetitive statement in Figure 9 with its statement $r := c$ **mod** d removed, the values of d, c, and q cannot change after the first, second, and third iterations, respectively.

Now we say that an expression e in S is *stable* if for every variable v such that $v \lambda_A e$, either $v \notin D_A$ or v is stable with respect to S. The *stability index* $\beta(e)$ of a

Input: (1) the set D_A of variables defined in a loop body A;
(2) *indegree*, an integer array in which, for each variable v in D_A, *indegree*[v] is the number of predecessors of v on $\tilde{R}_A$;
(3) for each variable in D_A, the set of its successors on $\tilde{R}_A$.
Output: (1) *stable*, a Boolean array in which, for each variable v in D_A, *stable*[v] = *true* if v is stable in the loop with body A and *stable*[v] = *false* otherwise;
(2) *index*, an integer array such that if v is stable then *index*[v] is the stability index of v.
Time complexity: $O(|D_A|^2)$.

```
begin
  S := φ;
  for each variable v in D_A do
    begin
      index[v] := 0;
      if indegree[v] = 0 then
        begin
          stable[v] := true;
          S := S ∪ {v}
        end
      else stable[v] := false
    end;
  while S ≠ φ do
    begin
      choose arbitrarily a variable w in S;
      S := S − {w};
      for each successor x of w on R̃_A do
        begin
          indegree[x] := indegree[x] − 1;
          index[x] := max(index[x], index[w] + 1);
          if indegree[x] = 0 then
            begin
              stable[x] := true;
              S := S ∪ {x}
            end
        end
    end
end
```

Fig. 10. Algorithm for finding stable variables and their stability indices.

stable expression e is defined by

$$\beta(e) = \begin{cases} 0, & \text{if } \{v \in D_A \mid v\lambda_A e\} = \phi \\ \max\{\alpha_S(v) \mid v \in D_A \text{ and } v\lambda_A e\} + 1, & \text{if } \{v \in D_A \mid v\lambda_A e\} \neq \phi. \end{cases}$$

It follows from the definition of α_S that, if an expression e in S is stable, then the evaluation of e will give the same result in iterations $\beta(e) + 1$, $\beta(e) + 2$, and all subsequent iterations. As an illustration, in the loop of Figure 9, without its statement $r := c$ **mod** d, the expressions 5, 6, 9, and 10 are stable, with stability indices 1, 2, 1, and 0, respectively.

Programmers sometimes deliberately construct repetitive statements in which some expressions are stable. However, we consider it important to detect and report the presence of all such expressions for the following reasons:

(1) If a Boolean expression defining a condition for exit from a loop is stable, this strongly suggests a programming error. For instance, in the case of a **while**

e **do** A statement, the stability of e implies that in any program execution at most $\beta(e)$ iterations can be performed, if the execution of the statement is to terminate. In particular, if $\beta(e) = 0$ the program certainly contains an error, and if $\beta(e) = 1$ then either the program contains an error or the repetitive statement could be replaced by **if** e **then** A. Even if $\beta(e) > 1$, the stability of e warrants attention, because some very common programming errors such as the omission of statements or misnaming of variables frequently manifest themselves in this way.

(2) The stability of one or more expressions within A suggests that either the repetitive statement S contains an error or that one or more statements could possibly be "hoisted" out of S. Such a modification often makes a program more comprehensible (and possibly more efficient). The detection and "hoisting" of assignment statements whose ⟨expression⟩ parts are stable of index 0 is well known in the context of optimizing compilers [2]. The techniques described here immediately reveal all the statements which could be treated in this way.

4.5 A Comparison with Data-Flow Analysis

The information given by the θ-, λ-, and μ-relations is somewhat similar in nature to that obtained by global data-flow analysis [3, 10–12, 15, 16]. Furthermore, our strategy for constructing flow relations, in traversing a parse tree, bears some resemblance to the data-flow methods based on graph grammars [8] and "high-level dataflow analysis" [20]. It is therefore instructive to relate these methods and, in particular, to compare the capabilities of our error-detection techniques with those based on data-flow analysis [9, 10, 13].

First, the relations $\tilde{\theta}$ and θ are easily interpreted in terms of "reaching definitions," as presented, for instance, by Hecht [12] and Kennedy [15]. Let us suppose that at the entry point of a program, we have an *initial definition* of each of its variables (with any variable which does not belong to V^i being assigned an "undefined value"). Then the condition $v\,\tilde{\theta}\,e$ signifies that the variable v appears in e, the initial definition of v reaches the statement whose ⟨expression⟩ part is e, and no other definition of v reaches that statement; whereas the condition $v\,\theta\,e$ signifies that v appears in e and the initial definition of v reaches the statement with ⟨expression⟩ part e. Thus the relations θ and $\tilde{\theta}$ indicate the "upward-exposed uses" of program variables; or in the terminology of Fosdick and Osterweil [9], for each variable $v \notin V^i$ the condition $v\,\theta\,e$ reveals a data-flow error in evaluating e, while the condition $v\,\tilde{\theta}\,e$ indicates a "significant" data-flow error.

The λ-relation is more difficult to interpret, because it provides information not obtainable by data-flow analysis, but we can view it as describing the "propagation" of upward-exposed uses of variables. (As an illustration, in our example of Section 4.3, whose relations are given in Figure 8, the λ-relation indicates that the initial (undefined) value of $a2$ may be used in evaluating the expressions 14, 15, and 16; these would all be *indirect* uses of the initial value of $a2$.) In fact, the $\tilde{\theta}$- and θ-relations are more useful for detecting references to undefined variables than the λ-relation, whose purpose is primarily to study information propagation in the manner described in Section 3.3.

The technique for finding ineffective statements from the μ-relation (described in Section 4.1) can be compared with methods of detecting "dead code," based on a live-variable analysis [12, 15]. The method based on the μ-relation is simpler

and more effective, because the μ-relation indicates directly all those statements (including conditional and repetitive statements) whose execution cannot affect the final values of any variables which are live on exit.

5. EXTENSIONS

The analysis methods of the previous sections can be extended by introducing relational models of procedure and function subprograms. In particular, a call of a procedure P can be represented in the following way. Let V^{i} be the set of actual parameters of P which are of mode *in* or *in out* and let V^{o} be the set of actual parameters of mode *in out* or *out*. Also, for each parameter $v \in V^{\mathrm{o}}$, let W_v be the set of members of V^{i} which *may be used* by P in obtaining the exit value of v. (The sets W_v could be obtained by a preliminary analysis of the procedure by the methods of the previous section, or they might be derived for instance from exit assertions for the procedure.) Then we may regard the procedure execution as a set of simultaneous assignments to the members of V^{o}, these assignments each being of the form $v := e_v$, where e_v is a fictitious expression with $\Gamma(e_v) = W_v$. Thus for the procedure call we have

$$D_S = V^{\mathrm{o}}, \qquad P_S = V - V^{\mathrm{o}}$$

$$\lambda_S = \bigcup_{v \in V^{\mathrm{o}}} (W_v \times \{e_v\}), \qquad \mu_S = \{(e_v, v) \mid v \in V^{\mathrm{o}}\}$$

$$\rho_S = \left(\bigcup_{v \in V^{\mathrm{o}}} (W_v \times \{v\}) \right) \cup (\iota - \{(v, v) \mid v \in V^{\mathrm{o}}\})$$

$$\theta_S = \bigcup_{v \in V^{\mathrm{o}}} (W_v \times \{e_v\}).$$

Function calls are easily accommodated if we ban side effects in functions. In this case a call of a function f in an expression e can be represented simply by including in $\Gamma(e)$ all the actual parameters of f.

Information-flow relations can also be derived for some further types of statements for sequence control by translating these into statements of the types treated in Section 2. The simplest example is the *case* statement, whose information-flow relations are easily derived by transforming it into nested conditional statements. As another example, the technique of "unrolling" loops which was applied in Section 2 to a **while**-statement can also be applied to **loop** statements with exits. For example, a **loop** statement with single-level exits of the general form

loop
  exit when e_1;
  A_1;
  exit when e_2;
  A_2;
  ⋮
  exit when e_k;
  A_k
end loop

has

$$D_S = \bigcup_{i=1}^{k} D_{A_i}, \qquad P_S = V$$

$$\lambda_S = R_k^* \bigcup_{i=1}^{k} (R_{i-1}\ (X_i \cup \lambda_{A_i}))$$

$$\rho_S = R_k^* \bigcup_{i=1}^{k} (R_{i-1}(\omega_i \cup \iota))$$

$$\mu_S = \bigcup_{i=1}^{k} (\psi_i \cup (M_{i-1} \cup M_k R_k^* R_{i-1})(\omega_i \cup \iota))$$

where

$$\chi_i = \Gamma(e_i) \times \{e_i\}, \qquad \psi_i = \{e_i\} \times D_S, \qquad \omega_i = \Gamma(e_i) \times D_S$$

and

$$R_i = \prod_{j=1}^{i} \rho_{A_j}, \qquad M_i = \bigcup_{j=1}^{i} \mu_j \left(\prod_{n=j+1}^{i} \rho_{A_n} \right)$$

are the ρ- and μ-relations of the sequence of statements $(A_1; A_2; \ldots; A_i)$.

6. PRACTICAL IMPLEMENTATION

The techniques described in this paper have been incorporated in SPADE, the Southampton Program Analysis and Development Environment. In this system, a number of program analysis and verification tools can be applied to a program represented in a functional description language (FDL). Translators to FDL have been developed, from a subset of Pascal, and from a subset of M6800 assembly code.

In the SPADE information-flow analyzer the flow relations are represented by list structures, because the relations are usually very sparse. Products of relations are constructed by the "classical" Boolean matrix multiplication algorithm, and transitive closures are obtained using Warshall's algorithm [22]; these algorithms are both of cubic worst case asymptotic time complexity. It would be possible to use algorithms with slightly lower time bounds: in particular, products could be formed by Strassen's method [21], and closures could be obtained for instance by Munro's algorithm [17], which embodies Strassen's method of multiplication. However, because the relations usually contain only a very small number of ordered pairs, we would not expect these more intricate algorithms to give a better practical performance.

From Table I it can be seen that, in using the classical multiplication and closure algorithms, our method of calculating the ρ-relation of a statement (given the ρ-relations of its constituents, if any) has a worst case asymptotic time complexity $O(|V|^3)$. Consequently, we have a time bound for calculating all the ρ-relations of a program of $O(|E| \times |V|^3)$. To obtain also the λ- and μ-relations, we have a time bound of $O(|E| \times |V|^2)$ for each statement, or $O(|E|^2 \times |V|^2)$ in all; to compute the θ and $\tilde{\theta}$ relations, the time bounds are $O(|E| \times |V|)$ for each statement and $O(|E|^2 \times |V|)$ in all. These bounds suggest that the time

requirements could be severe, but in practice, because the relations are extremely sparse, the running times have been found to be quite acceptable: for instance, in a Pascal implementation on a PDP-11/44, the information-flow analysis of the covariance procedure 2 of Witten [24], which has 14 variables and 50 statements, of which 15 are **for**-statements requiring closure computations, took 8 seconds.

With regard to space complexity, in the worst case the space required to store a single ρ-relation is of order $O(|V|^2)$, and therefore (in the worst case) the total space required in computing the ρ-relation for a program is $O(|E| \times |V|^2)$. To compute as well the λ-, μ-, and θ-relations the additional space requirements, in the worst case, are of order $O(|E|^2 \times V)$. Again, in practice the space requirements have been found to be much less severe than these bounds would suggest, partly because the relations are very sparse, and also because in following the procedure described in Section 2.3, the number of relations which need to be retained in analyzing a complete program is relatively small.

The information-flow analyzer is provided with specifications of the sets V^i and V^o of variables whose values are imported and exported by a program or subprogram. (These can be passed to it, for instance, via "annotations" of a Pascal text.) This allows the analyzer to detect from the computed relations all errors of the kinds discussed in Section 4, which are signaled to the SPADE user. Stability tests on each repetitive statement are performed at the time when its ρ-relation is constructed.

7. CONCLUSIONS

The calculation of the information-flow relations is feasible, in polynomial time and space, for an important class of programs. Although the analysis is purely syntactic, with the underlying assumption that all program paths are executable, the methods of extracting partial statements and of finding statements that may affect or may be affected by particular variable values within a program can greatly assist one's comprehension of its action. These methods are currently being incorporated in a program development environment, in which it will be possible to use the μ- and λ-relations to determine which program statements are to be displayed by a text editor. The ρ-relation also provides a useful indication of whether the relationship between input and output variables is as required.

With regard to automatic error detection, the tests for ineffectiveness of statements and variables and for loop stability are easy to perform and usefully extend the class of errors and anomalies which can be detected by static analysis.

ACKNOWLEDGMENTS

The authors are indebted to Dr. B. D. Bramson, Dr. W. J. Cullyer, and S. J. Goodenough of the Royal Signals and Radar Establishment, Malvern, for helpful discussions.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. AHO, A.V., AND ULLMAN, J.D. *The Theory of Parsing, Translation and Compiling*, Vol. 2: Compiling. Prentice-Hall, Englewood Cliffs, N.J., 1973.

3. ALLEN, F.E., AND COCKE, J. A program data flow analysis procedure. *Commun. ACM 19*, 3 (Mar. 1976), 137–147.
4. ANDREWS, G.R., AND REITMAN, R.P. An axiomatic approach to information flow in programs. *ACM Trans. Prog. Lang. Syst. 2*, 1 (Jan. 1980), 56–76.
5. CARRÉ, B.A. *Graphs and Networks.* Oxford University Press, New York, 1979.
6. COHEN, E. Information transmission in sequential programs. In *Foundations of Secure Computation*, R. A. Demillo et al., Ed. Academic Press, New York, 1978, pp. 297–335.
7. DENNING, D.E., AND DENNING, P.J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (July 1977), 504–513.
8. FARROW, R., KENNEDY, K., AND ZUCCONI, L. Graph grammars and global program flow analysis. In *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science* (Houston, Tex., Nov.). IEEE, New York, 1975, pp. 42–56.
9. FOSDICK L.D., AND OSTERWEIL, L.J. Validation and global optimization of programs. In *Proceedings of the 4th Texas Conference on Computing Systems* (Austin, Tex.). 1975. Sponsored by the IEEE Computer Society.
10. FOSDICK, L.D., AND OSTERWEIL, L.J. Data flow analysis in software reliability. *ACM Comput. Surv. 8*, 3 (Sept. 1976), 305–330.
11. GRAHAM, S.L., AND WEGMAN, M. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*, 1 (Sept. 1976), 172–202.
12. HECHT, M.S. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.
13. HUANG, J.C. Detection of data flow anomaly through program instrumentation. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 226–236.
14. JENSEN, K., AND WIRTH, N. *PASCAL User Manual and Report*, 2nd ed. Springer-Verlag, New York, 1974.
15. KENNEDY, K. A survey of data flow analysis techniques. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 5–54.
16. KILDALL, G.A. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Boston, Mass., Oct.). ACM, New York, 1973, pp. 194–206.
17. MUNRO, I. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett. 1* (1971), 56–58.
18. OSTERWEIL, L.J. Using data flow tools in software engineering. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 237–263.
19. POPEK, G.J., HORNING, J.J., LAMPSON, B.W., MITCHELL, W.G., AND LONDON, R.L. Notes on the Design of EUCLID. In *SIGPLAN Not. 12*, 3 (Mar. 1977), 11–18.
20. ROSEN, B.K. High level data flow analysis. *Commun. ACM 20*, 10 (Oct. 1977), 712–724.
21. STRASSEN, V. Gaussian elimination is not optimal. *Numer. Math. 13* (1969).
22. WARSHALL, S. A theorem on Boolean matrices. *J. ACM 9*, 1 (Jan. 1962), 11–13.
23. WERSER, M. Programmers use slicing when debugging. *Commun. ACM 25*, 7 (July 1982), 446–452.
24. WITTEN, I.H. Algorithms for adaptive linear prediction. *Comput. J. 23*, 1 (Feb. 1980), 78–84.

Received April 1982; revised August 1983 and July 1984; accepted July 1984