A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance

ANDY PODGURSKI AND LORI A. CLARKE

Abstract—**A formal, general model of program dependences is presented and used to evaluate several dependence-based software testing, debugging, and maintenance techniques. Two generalizations of control and data flow dependence, called weak and strong syntactic dependence, are introduced and related to a concept called semantic dependence. Semantic dependence models the ability of a program statement to affect the execution behavior of other statements. It is shown, among other things, that weak syntactic dependence is a necessary but not sufficient condition for semantic dependence and that strong syntactic dependence is a necessary but not sufficient condition for a restricted form of semantic dependence that is finitely demonstrated. These results are then used to support some proposed uses of program dependences, to controvert others, and to suggest new uses.**

Index Terms—**Data flow testing, program analysis, program dependences, program slicing, software debugging, software maintenance, software testing.**

I. INTRODUCTION

PROGRAM dependences are relationships, holding between program statements, that can be determined from a program's text and used to predict aspects of the program's execution behavior. There are two basic types of program dependences: "control dependences," which are features of a program's control structure, and "data flow dependences," which are features of a program's use of variables. Informally, a statement s is control dependent on the branch condition c of a conditional branch statement if the control structure of the program indicates that c potentially decides, via the branches it controls, whether s is executed or not. For example, in the program of Fig. 1, statements 3 and 4 are control dependent on the branch condition at line 2. Informally, a statement s is data flow dependent on a statement s' if data potentially propagates from s' to s via a sequence of variable assignments. For example, in the program of Fig. 1, statement 5 is data flow dependent on statement 1, since data potentially propagates from statement 1 to statement 5. *Dependence analysis*, the process of determining a program's dependences, combines traditional control flow analysis and data flow analysis [2], and hence can be implemented efficiently.

Until recently, most proposed uses of program dependences have been justified only informally, if at all. Since program dependences are used for such critical purposes as software testing [15], [16], [19], [22], debugging [3], [28], and maintenance [23], [28], code optimization and parallelization [8], [20], and computer security [7],[1] this informality is risky. This paper supplements other recent investigations of the semantic basis for the uses of program dependences [4], [13], [25] by presenting a formal, general model of program dependences and by using it to evaluate several dependence-based software testing, debugging, and maintenance techniques. The results support certain proposed uses of program dependences, controvert others, and suggest new ones.

One example of our results involves the use of program dependences to find "operator faults" in programs. An *operator fault* is the presence of an inappropriate operator[2] in a program statement. For instance, accidental use of the multiplication operator "*" instead of the addition operator "+" in the assignment statement "**X := Y * Z**" results in an operator fault. It would be useful to be able to automatically detect and locate operator faults; unfortunately, as with many other semantic questions about programs, the question of whether a program contains an operator fault or not is undecidable. This paper shows, however, that there is an algorithm, in fact an *efficient* one, that detects *necessary conditions* for an operator fault at one statement to affect the execution behavior of another statement. These necessary conditions are expressed in terms of program dependences. Consequently, dependences can be used to help locate the statements that might be affected by an operator fault at a given statement.

To determine some of the implications of program dependences, we relate control and data dependence to a concept called "semantic dependence." Informally, a program statement s is semantically dependent on a statement s' if the semantics of s', that is, the function computed by s', potentially affects the execution behavior of s. The significance of semantic dependence is that it is a

Manuscript received October 15, 1989; revised May 1, 1990. Recommended by N. G. Leveson.

A. Podgurski is with the Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH 44106.

L. A. Clarke is with the Software Development Laboratory, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

IEEE Log Number 9037077.

[1]The term "dependence" is not used in all the references given.

[2]The term "operator" refers to both the predefined operators of a programming language and to user-defined procedures and functions.

0098-5589/90/0900-0965$01.00 © 1990 IEEE

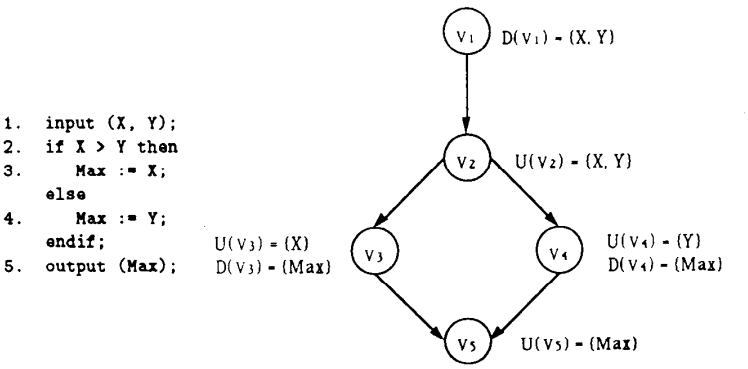


Fig. 1. Max program and its def/use graph.

necessary condition for certain interstatement semantic relationships. For example, if s and s' are distinct statements, then s must be semantically dependent on s' for an operator fault at s' to affect the execution behavior of s. Similarly, some output statement must be semantically dependent on a statement s for the semantics of s to affect the output of a program.

Three main results are presented in this paper:

1) A generalization of control and data dependence, called "weak syntactic dependence," is a necessary condition for semantic dependence.

2) A commonly used generalization of control and data dependence, which we call "strong syntactic dependence," is a necessary condition for semantic dependence only if the semantic dependence does not depend in a certain way on a program failing to terminate.

3) Neither data flow, weak syntactic, nor strong syntactic dependence is a sufficient condition for semantic dependence.

We use these results to evaluate several dependence-based testing, debugging, and maintenance techniques.

Section II defines some necessary terminology and Section III defines control, data, and syntactic dependence. Semantic dependence is informally defined in Section IV and then related to syntactic dependence in Section V. In Section VI, the implications of the results of Section V for software testing, debugging, and maintenance are described. In Section VII, related work not already considered in Section VI is surveyed. Section VIII presents a summary and discussion of possible future research directions. In the Appendix, the formal definition of semantic dependence is presented, and the proofs of the two most significant results of Section V are sketched.

II. Terminology

In this section we define control flow graphs, some dominance relations, and def/use graphs.

A *directed graph or digraph* G is a pair $(V(G), A(G))$, where $V(G)$ is any finite set and $A(G)$ is a subset of $V(G) \times V(G) - \{(v, v) \mid v \in V(G)\}$. The elements of $V(G)$ are called *vertices* and the elements of $A(G)$ are called *arcs*. If $(u, v) \in A(G)$ then u is *adjacent to* v and v is *adjacent from* u; the arc (u, v) is *incident to* v and *incident from* u. A *predecessor* of a vertex v is a vertex adjacent to v, and a *successor* of v is a vertex adjacent from v. The *indegree* of a vertex v is the number of predecessors of v, and the *outdegree* of v is the number of successors of v.

A *walk* W in G is a sequence of vertices $v_1\, v_2 \cdots v_n$ such that $n \geq 0$ and $(v_i, v_{i+1}) \in A(G)$ for $i = 1, 2, \cdots, n-1$. The *length* of a walk $W = v_1 v_2 \cdots v_n$, denoted $|W|$, is the number n of vertex occurrences in W. Note that a walk of length zero has no vertex occurrences; such a walk is called *empty*. A nonempty walk whose first vertex is u and whose last vertex is v is called a $u-v$ *walk*. If $W = w_1 w_2 \cdots w_m$ and $X = x_1 x_2 \cdots x_n$ are walks such that either W is empty, X is empty, or w_m is adjacent to x_1, then the *concatenation* of W and X, denoted WX, is the walk $w_1 w_2 \cdots w_m x_1 x_2 \cdots x_n$.

All the types of dependence considered in this paper are directly or indirectly defined in terms of a "control flow graph," which represents the flow of control in a sequential, procedural program.

Definition 1: A *control-flow graph* G is a directed graph that satisfies each of the following conditions:

1) The maximum outdegree of the vertices of G is at most two[3].

2) G contains two distinguished vertices: the *initial vertex* v_I, which has indegree zero, and the *final vertex* v_F, which has outdegree zero.

3) Every vertex of G occurs on some $v_I - v_F$ walk.

A vertex of outdegree two in a control flow graph is called a *decision vertex*, and an arc incident from a decision vertex is called a *decision arc*. The set of decision vertices of G is denoted $V_{dec}(G)$.

Here, the vertices of a control flow graph represent simple program statements (such as assignment statements and procedure calls) and also branch conditions, while the arcs represent possible transfers of control between these. The program's entry point and exit point are represented by the initial vertex and final vertex, respectively. A de-

[3]This restriction is made for simplicity only.

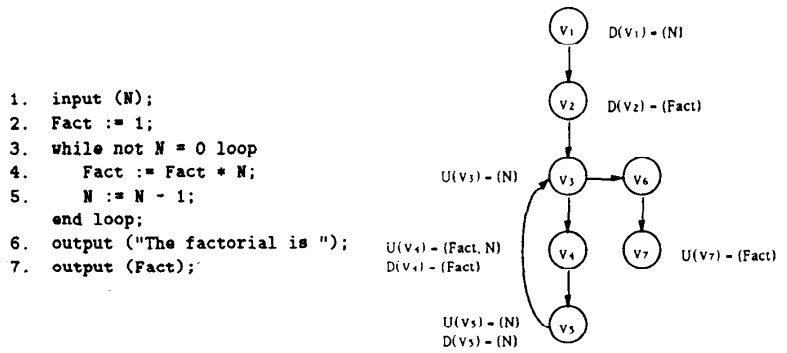


Fig. 2. Factorial program and its def/use graph.

cision vertex represents the branch condition of a conditional branch statement. The definition given here of a control flow graph is somewhat restricted, to simplify the presentation of results. This definition can be used to represent any procedural program, however, by employing straightforward representation conventions involving the use of dummy vertices and arcs.

The control flow graph of the program in Fig. 1 is shown alongside the program; the annotations to this graph are explained subsequently.

The next three definitions are used in defining types of control dependence.

Definition 2: Let G be a control flow graph. A vertex $u \in V(G)$ *forward dominates* a vertex $v \in V(G)$ iff every $v-v_F$ walk in G contains u; u *properly forward dominates* v iff $u \neq v$ and u forward dominates v.

Definition 3: Let G be a control flow graph. A vertex $u \in V(G)$ *strongly forward dominates* a vertex $v \in V(G)$ iff u forward dominates v and there is an integer $k \geq 1$ such that every walk in G beginning with v and of length $\geq k$ contains u.

In the control flow graph of Fig. 1, v_5 (strongly) forward dominates each vertex, whereas v_3 and v_4 forward dominate only themselves. In the control flow graph of Fig. 2, v_5 strongly forward dominates v_4, but v_6 does not strongly forward dominate v_4, because there are arbitrarily long walks from v_4 that do not contain v_6.

While control dependence has been defined in terms of forward dominance before [7], [8], the use of strong forward dominance for this purpose is apparently new.

We state the following theorem without proof.

Theorem 1: Let G be a control flow graph. For each vertex $u \in (V(G) - \{v_F\})$, there exists a proper forward dominator v of u such that v is the first proper forward dominator of u to occur on every $u-v_F$ walk in G.

The "immediate forward dominator" of a decision vertex d is the vertex where all walks leaving d first come together again. More formally:

Definition 4: Let G be a control flow graph. The *immediate forward dominator* of a vertex $v \in (V(G) - \{v_F\})$, denoted $ifd(v)$, is the vertex that is the first proper forward dominator of v to occur on every $v-v_F$ walk in G.

For example, in the control flow graph of Fig. 1, v_5 is the immediate forward dominator of v_2, v_3, and v_4. In the control flow graph of Fig. 2, v_6 is the immediate forward dominator of v_3.

Data, syntactic, and semantic dependence are defined in terms of an annotated control flow graph called a "def/use graph." For each vertex v in a def/use graph, $D(v)$ denotes the set of variables defined (assigned a value) at the statement represented by v, and $U(v)$ denotes the set of variables used (having their values referenced) at that statement. A def/use graph is similar to a program schema [11], [17] and is essentially the program representation used in data flow analysis [2].

Definition 5: A *def/use graph* is a quadruple $G = (G, \Sigma, D, U)$, where G is a control flow graph, Σ is a finite set of symbols called *variables*, and D: $V(G) \rightarrow \mathcal{P}(\Sigma)$, U: $V(G) \rightarrow \mathcal{P}(\Sigma)$ are functions.[4]

The def/use graphs of the programs in Figs. 1 and 2 are shown alongside the programs.

Definition 6: Let $G = (G, \Sigma, D, U)$ be a def/use graph, and let W be a walk in G. Then

$$D(W) = \bigcup_{v \in W} D(v).$$

For example, referring to the def/use graph of Fig. 1, $D(v_1 v_2 v_3 v_5) = \{\textbf{Max, X, Y}\}$.

As usual in the static analysis of programs, exactly what constitutes a variable, definition, or use is sometimes a subtle issue [2]. In the model of computation we adopt (see Section IV), values are associated with the variables (names) in Σ, and a vertex v can interrogate only the values of variables in $U(v)$ and modify only the values of variables in $D(v)$. In representing a program with our formalism, this means that if the statement represented by a vertex v reads or writes a storage element, or if this is *uncertain*, then a variable name that denotes the potentially accessed storage element should be included in $U(v)$ or $D(v)$, respectively. When the accessed storage element cannot be determined until runtime, it is permissible to include either a distinct name for each storage element that might be accessed or one name denoting all

[4]We denote the power set (set of all subsets) of a set S by $\mathcal{P}(S)$.

such elements, provided the same representation is used consistently. When the same storage element is accessed via different names in the program, all of these names can be included or a single name can be used in place of them. These are the conventions usually adopted in static analysis, where, for example, an access through a pointer is often treated as an access to all the objects that might be pointed to, and an access to an array element is often treated as an access to the entire array. These conventions are conservative, in that they may indicate data flow that cannot actually occur, but are safe, in that they will not fail to indicate any data flow that does occur.

III. Control, Data, and Syntactic Dependence

A. Control Dependence

The concept of control dependence is used to model the effect of conditional branch statements on the behavior of programs. Control dependence is a property of a program's control structure alone, in that it can be defined strictly in terms of a control flow graph. Various formal and informal definitions of control dependence are given in the literature. Usually these are expressed in terms of "structured" control statements of a particular language or class of languages. Such definitions have limited applicability, because control statements vary across languages and because "unstructured" programs occur in practice. Indeed, even judicious use of the *goto* statement or the use of restricted branch statements such as Ada's *exit*, *raise*, and *return* statements can result in programs that are, strictly speaking, unstructured. It is therefore desirable to have a language-independent definition of control dependence that applies to both structured and unstructured programs. Two definitions that satisfy these requirements are those of "weak control dependence" and "strong control dependence."

Strong control dependence was originally defined in the context of computer security [7][5], and this definition has been used by several authors [15], [20], [28]. To our knowledge, it was the first graph-theoretic, language and structure-independent characterization of control dependence to appear in the literature.

Definition 7: Let G be a control flow graph, and let u, $v \in V(G)$. Then u is *strongly control dependent* on v iff there exists a $v-u$ walk vWu not containing the immediate forward dominator of v.

For example, in the control flow graph of Fig. 1, the immediate forward dominator of the decision vertex v_2 is v_5; therefore v_3 and v_4 are strongly control dependent on v_2. In the control flow graph of Fig. 2, the immediate forward dominator of the decision vertex v_3 is v_6; therefore v_3, v_4, and v_5 are strongly control dependent on v_3. Note that the statements that are strongly control dependent on the branch condition of a structured *if-then* or *if-then-else* statement are those in its "body." The statements that are strongly control dependent on the branch condition of a structured *while* or *repeat-until* loop are the branch condition itself and the statements in the loop's body.

Weak control dependence [21] is a generalization of strong control dependence in the sense that every strong control dependence is also a weak control dependence.

Definition 8: Let G be a control flow graph, and let u, $v \in V(G)$. Vertex u is *directly weakly control dependent* on vertex v iff v has successors v' and v'' such that u strongly forward dominates v' but does not strongly forward dominate v''; u is *weakly control dependent* on v iff there is a sequence $v_1, v_2, \cdots, v_n$ of vertices, $n \geq 2$, such that $u = v_1$, $v = v_n$, and v_i is directly weakly control dependent on v_{i+1} for $i = 1, 2, \cdots, n-1$.

Informally, u is directly weakly control dependent on v if v has successors v' and v'' such that if the branch from v to v' is executed then u is necessarily executed within a fixed number of steps, while if the branch from v to v'' is taken then u can be bypassed or its execution can be delayed indefinitely.

The essential difference between weak and strong control dependence is that weak control dependence reflects a dependence between an exit condition of a loop and a statement outside the loop that may be executed after the loop is exited, while strong control dependence does not. For example, in the control flow graph of Fig. 2, v_6 is (directly) weakly control dependent on v_3 (because v_6 strongly forward dominates itself, but not v_4), but not strongly control dependent on v_3 (because v_6 is the immediate forward dominator of v_3). In addition, v_3, v_4, and v_5 are (directly) weakly control dependent on v_3, because each strongly forward dominates v_4 but not v_6. The additional dependences of the weak control dependence relation are relevant to program behavior, because an exit condition of a loop potentially determines whether execution of the loop terminates.

The weak and strong control dependence relations for a control flow graph G can be computed in $O(|V(G)|^3)$ time [21].

B. Data Flow Dependence

Although several other types of data dependence are discussed in the literature, we require only data flow dependence [8], [20].

Definition 9: Let $G = (G, \Sigma, D, U)$ be a def/use graph, and let u, $v \in V(G)$. Vertex u is *directly data flow dependent* on vertex v iff there is a walk vWu in G such that $(D(v) \cap U(u)) - D(W) \neq \varnothing$; u is *data flow dependent* on v iff there is a sequence $v_1, v_2, \cdots, v_n$ of vertices, $n \geq 2$, such that $u = v_1$, $v = v_n$ and v_i is directly data flow dependent on v_{i+1} for $i = 1, 2, \cdots, n-1$.

Note that if u is data flow dependent on v then there is a walk $v_1 W_1 v_2 W_2 \cdots v_{n-1} W_{n-1} v_n$, $n \geq 2$, such that $v = v_1$, $u = v_n$, and $(D(v_i) \cap U(v_{i+1})) - D(W_i) \neq \varnothing$ for $i = 1, 2, \cdots, n-1$. Such a walk is said to *demonstrate* the data flow dependence of u upon v.

Referring to the def/use graph of Fig. 1, v_3 is directly data flow dependent on v_1, because the variable **X** is defined at v_1, used at v_3, and not redefined along the walk

[5] In [7] the concept is called "implicit information flow."

$v_1 v_2 v_3$; v_5 is directly data flow dependent on v_3, because the variable **Max** is defined at v_3, used at v_5, and not redefined along the walk $v_3 v_5$. It follows that v_5 is data flow dependent on v_1; the walk $v_1 v_2 v_3 v_5$ demonstrates this dependence.

The direct data flow dependence relation for a control flow graph can be computed efficiently using a fast algorithm for the "reaching definitions" problem [2]. The data flow dependence relation can then be efficiently computed using a fast algorithm for transitive closure [1].

C. Syntactic Dependence

To evaluate uses of control and data dependence, it is necessary to consider *chains* of such dependences, that is, sequences of vertices in which each vertex except the last is either control dependent or data dependent on the next vertex. Informally, there is a "*weak* syntactic dependence" between two statements if there is a chain of data flow and/or *weak* control dependences between the statements, while there is a "*strong* syntactic dependence" between the statements if there is a chain of data flow dependences and/or *strong* control dependences between them. Weak syntactic dependence apparently has not been considered before in the literature; the notion of strong syntactic dependence is implicit in the work of several authors [3], [7], [8], [13], [15], [20], [28].

Definition 10: Let $G = (G, \Sigma, D, U)$ be a def/use graph, and let $u, v \in V(G)$. Vertex u is *weakly syntactically dependent* (*strongly syntactically dependent*) on vertex v iff there is a sequence $v_1, v_2, \cdots, v_n$ of vertices, $n \geq 2$, such that $u = v_1$, $v = v_n$, and for $i = 1, 2, \cdots, n-1$, either v_i is weakly control dependent (strongly control dependent) on v_{i+1} or v_i is data flow dependent on v_{i+1}.

Since the weak and strong control dependence and data flow dependence relations for a def/use graph can be computed efficiently, the weak and strong syntactic dependence relations can be computed efficiently by using a fast algorithm for transitive closure.

Referring to the def/use graph of Fig. 2, v_6 is weakly syntactically dependent on v_5, because v_6 is weakly control dependent on v_3 and v_3 is data flow dependent on v_5; v_5 is strongly syntactically dependent on v_1, because v_5 is strongly control dependent on v_3 and v_3 is data flow dependent on v_1. Note that v_6 is not strongly syntactically dependent on v_5.

IV. Semantic Dependence

Recall that, informally, a statement s is semantically dependent on a statement s' if the function computed by s' affects the execution behavior of s. In this section, a more precise but still informal description of semantic dependence is given. The formal definition is presented in the Appendix.

We first informally define the auxiliary terms necessary to define semantic dependence. A sequential procedural program can be viewed abstractly as an *interpreted* def/use graph. An *interpretation* of a def/use graph is an assignment of partial computable functions to the vertices of the graph. The function assigned to a vertex v is the one computed by the program statement that v represents; it maps values for the variables in $U(v)$ to values for the variables in $D(v)$ or, if v is a decision vertex, to a successor of v. An interpretation of a def/use graph is similar to an interpretation of a program schema [11], [17]. An operational semantics for interpreted def/use graphs is defined in the obvious way, with computation proceeding sequentially from vertex to vertex along the arcs of the graph, as determined by the functions assigned to the vertices. A *computation sequence* of a program is the sequence of states (pairs consisting of a vertex and a function assigning values to all the variables in the program) induced by executing the program with a particular input. An *execution history* of a vertex v is the sequence whose ith element is the assignment of values held by the variables of $U(v)$ just before the ith time v is visited during a computation. An execution history of a vertex is an interpreted def/use graph abstracts the "execution behavior" of a program statement.

A more precise description of semantic dependence can now be given.

Definition 11 (Informal): A vertex u in a def/use graph G is *semantically dependent* on a vertex v of G if there are interpretations I_1 and I_2 of G that differ only in the function assigned to v, such that for some input, the execution history of u induced by I_1 differs from that induced by I_2.[6]

For example, if the branch condition **X > Y** in the program of Fig. 1 were changed to **X < Y**, then the program would compute the *Min* function instead of the *Max* function. Hence, for all unequal values of **X** and **Y**, this change demonstrates that vertex v_5 of the def/use graph of Fig. 1 is semantically dependent on vertex v_2. As another example, if the statement **N := N − 1** in the program of Fig. 2 were changed to **N := N − 2**, the *while*-loop would fail to terminate for the input **N** = 5, preventing statement 6 from executing. Hence, this change demonstrates that vertex v_6 of the def/use graph of Fig. 2 is semantically dependent on vertex v_5.

Note that a pair of execution histories that demonstrate a semantic dependence can differ in two ways: a) the histories have corresponding entries that are unequal and b) one history is longer than the other. Informally, the semantic dependence is said to be *finitely demonstrated* if either:

1) Condition a) holds; or

2) Condition b) is demonstrated by finite portions of the computation sequences that caused the execution histories.

Semantic dependence demonstrated by a pair of *halting* computations is, of course, finitely demonstrated. For ex-

[6]The formal definition of semantic dependence, given in the Appendix, contains conditions to ensure that a semantic dependence is not caused by the value of the function assigned to a vertex being undefined for some input. This is done to avoid trivial semantic dependences. When we informally refer to (the semantics of) one program statement affecting the execution behavior of another statement, this restriction is implied.

ample, the semantic dependence of vertex v_5 upon vertex v_2 in the def/use graph of Fig. 1 is finitely demonstrated, because the *Min* and *Max* functions are defined for all pairs of integers. The semantic dependence of vertex v_6 upon vertex v_5 in the def/use graph of Fig. 2 is not finitely demonstrated, because the only way v_5 can affect the execution behavior of v_6 is by determining whether execution of the cycle $v_3 v_4 v_5 v_3$ terminates. Note that even *nonterminating* computations can finitely demonstrate semantic dependence, via their finite initial segments. For example, despite the fact that it makes the program fail to terminate, changing the statement **N := N − 1** in the program of Fig. 2 to **N := N − 2** finitely demonstrates that vertex v_5 of the programs' def/use graph is semantically dependent on itself, because the change alters the argument to the second execution of statement 5 for the input **N** = 5.

V. Relating Semantic and Syntactic Dependence

In software testing, debugging, and maintenance, one is often interested in the following question:

> When can a change in the semantics of a program statement affect the execution behavior of another statement?

This question is, however, undecidable in general. Dependence analysis, like data flow analysis, avoids problems of undecidability by trading precision for (efficient) decidability. During dependence analysis, programs are represented by def/use graphs, which contain limited semantic information but are easily analyzed. Dependence analysis allows semantic questions to be answered "approximately," because a program's dependences partially determine its semantic properties. To evaluate the usefulness of dependence analysis in "approximately" answering the question above, we frame the question in terms of def/use graphs, by asking "When is one statement semantically dependent on another?". This leads to our main results.

(The proofs of Theorems 2 and 4 below are sketched in the Appendix. The proofs of Theorems 3 and 5 are given informally with the theorems. Formal versions of all these proofs are found in [21].)

Theorem 2: Let $G = (G, \Sigma, D, U)$ be a def/use graph, and let $u, v \in V(G)$. If u is semantically dependent on v then u is weakly syntactically dependent on v.

It was shown in Section IV that vertex v_6 in the def/use graph of Fig. 2 is semantically dependent on vertex v_5. This is reflected by the fact that v_6 is weakly syntactically dependent on v_5, as shown in Section III-C. However, v_6 is not strongly syntactically dependent on v_5.

Theorem 3: Strong syntactic dependence is not a necessary condition for semantic dependence.

The next theorem shows that strong syntactic dependence does have semantic significance. The theorem in fact justifies some informally-posed applications of dependence analysis (see Section VI).

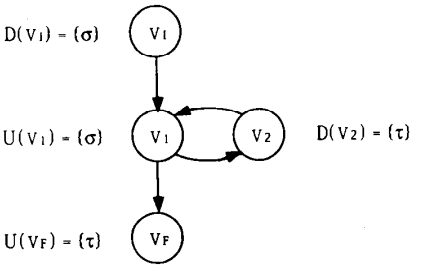


Fig. 3. Direct data flow dependence without semantic dependence.

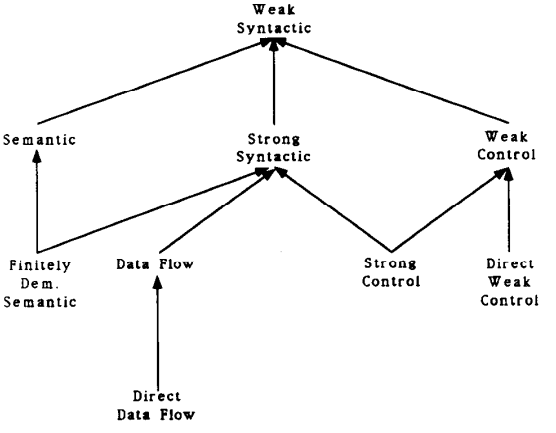


Fig. 4. Relationships of dependence types.

Theorem 4: Let $G = (G, \Sigma, D, U)$ be a def/use graph, and let $u, v \in V(G)$. If u is semantically dependent on v and this semantic dependence is finitely demonstrated then u is strongly syntactically dependent on v.

It was shown in Section IV that vertex v_5 in the def/use graph of Fig. 1 is semantically dependent on vertex v_2 and that this semantic dependence is finitely demonstrated. This is reflected by the fact that v_5 is strongly syntactically dependent on v_2 (v_5 is directly data flow dependent on v_3, which is strongly control dependent on v_2).

Theorem 5: Neither direct data flow dependence nor data flow dependence is a sufficient condition for semantic dependence.

In the def/use graph of Fig. 3, vertex v_F is directly data flow dependent on vertex v_2. However, v_F is not semantically dependent on v_2. This is because no computation of any program with this def/use graph visits both v_2 and v_F, since the "loop control variable" σ is not redefined in the cycle $v_1 v_2 v_1$.

Corollary 1: Neither weak nor strong syntactic dependence is a sufficient condition for semantic dependence.

Fig. 4 shows the dependence relations considered in this paper, ordered by set inclusion. There is an arrow from a relation R_1 to a relation R_2 if $R_1 \subseteq R_2$.

VI. Implications of the Results

The results of Section V support the following general conclusions about the use of dependence analysis to obtain information about relationships between program statements:

1) The absence of weak syntactic dependence between two statements precludes all relationships between them that imply semantic dependence.

2) The absence of strong syntactic dependence between two statements does not necessarily preclude all relationships between them that imply (nonfinitely-demonstrated) semantic dependence.

3) The absence of strong syntactic dependence between two statements precludes all relationships between them that imply finitely demonstrated semantic dependence.

4) The presence of direct data flow dependence, data flow dependence, or weak or strong syntactic dependence between two statements does not necessarily indicate any relationship between the statements that implies semantic dependence.

Conclusion 1 follows from Theorem 2; any relationship between two statements that implies semantic dependence also implies weak syntactic dependence. Conclusion 2 follows from Theorem 3 and Theorem 4; nonfinitely-demonstrated semantic dependence does not imply strong syntactic dependence. Conclusion 3 follows from Theorem 4; any interstatement relationship that implies finitely demonstrated semantic dependence also implies strong syntactic dependence. Finally, conclusion 4 follows from Theorem 5 and Corollary 1.

Note that conclusion 1 implies that the weak syntactic dependence relation for a program is an "upper bound" for (contains) any relation on the program's statements that implies semantic dependence. Similarly, conclusion 3 implies that the strong syntactic dependence relation for a program bounds any relation on the program's statements that implies finitely demonstrated semantic dependence. Since the syntactic dependences in a program can be computed efficiently, these bounds can be determined easily and used to narrow the search for statements having certain important relationships. For example, if an operator fault at a statement s affects the execution behavior of a statement s', this demonstrates that s' is semantically dependent on s; therefore, only those statements that are weakly syntactically dependent on s could be affected by an operator fault at s. Consequently, weak syntactic dependences can be used to help locate statements that can be affected by an operator fault at a given statement. Of course, whenever a relation R on the statements of a program implies finitely demonstrated semantic dependence, the strong syntactic dependence relation for the program is a "tighter" bound on R than the weak syntactic dependence relation is.

In the sequel, we use the results of Section V to evaluate the semantic basis for several proposed uses of dependences in testing, debugging, and maintenance and to suggest new uses. The results suggest that some proposed uses are mistaken, but provide partial justification, in terms of facilitating search, for other uses.

A. Dependence-Coverage Criteria

In software testing, a *dependence-coverage criterion* is a test-data selection rule based on "covering" or "exercising" certain program dependences. Several coverage criteria have been defined that call for exercising the data flow dependences in a program [10], [16], [19], [22]; these are called "data flow coverage criteria." They require the execution of program walks[7] that demonstrate certain data flow dependences. One rationale for the data flow coverage criteria is that they facilitate the detection of incorrect variable definitions [16], [22], which may be caused by mistaken use of operators and/or variable names. Another is that they facilitate the detection of faults that cause missing and spurious data flow dependences [16]. These two rationales are related, since an incorrect variable definition can cause missing or spurious data flow dependences. In Section VI-A-1 below we use the results of Section V to evaluate the use of dependence coverage criteria for the detection of those incorrect variable definitions that arise from operator faults. In Section VI-A-2, we use the results of Section V to evaluate the use of dependence coverage criteria for the detection of faults that cause missing and spurious dependences.

1) Detection of Operator Faults: The data flow coverage criteria address operator faults by exercising data flow dependences upon potentially faulty variable definitions. This may cause erroneous values produced by an operator fault to propagate, via the sequence of assignments represented by a data flow dependence, and produce an observable failure. The data flow coverage criteria differ with regard to the number and type of data flow dependences exercised and with regard to the number of walks executed that demonstrate a given dependence. For example:

- Rapps and Weyuker's All-Defs criterion [22] exercises one direct data flow dependence, if possible, upon each variable definition in a program.
- Rapps and Weyuker's All-Uses and All-DU-Paths criteria [22] exercise all direct data flow dependences in a program.
- Laski and Korel's Strategy II [16] exercises all direct data flow dependences, but exercises them in combination.
- For a fixed k, Ntafos's Required k-Tuples criterion [19] exercises all chains of k direct data flow dependences; that is, all sequences of $k + 1$ vertices such that each vertex in the sequence except the last is directly data flow dependent on the next.

[7] That is, sequences of statements corresponding to walks in a program's control flow graph.

It is plausible that the propagation of erroneous values via data flow dependences alone is sufficiently common to make data flow coverage criteria worthwhile, even though, by conclusion 4, that fact that a statement s is data flow dependent on a faulty statement s' does not imply that an erroneous value propagates from s' to s. However, when an operator fault causes a conditional statement to make an incorrect branch, *control dependence* can be critical to the ability of the fault to affect the execution behavior of a particular statement. For example, changing the branch predicate $\mathbf{X} > \mathbf{Y}$ in the program of Fig. 1 to $\mathbf{X} < \mathbf{Y}$ changes the number of times statement 3 is executed for the inputs $\mathbf{X} = 5$, $\mathbf{Y} = 2$ because statement 3 is strongly control dependent on statement 2. Hence, no coverage criterion based only on exercising data flow dependences can, in general, exercise all the syntactic dependences associated with erroneous information flow produced by operator faults. Correspondingly, it is easily seen that data flow dependence is not a necessary condition for even finitely demonstrated semantic dependence.

Some of the data flow coverage criteria mentioned above, such as the All-Uses and Required k-Tuples criteria, incorporate limited forms of control dependence coverage. However, even these criteria do not exercise all syntactic dependences associated with the erroneous information flow produced by operator faults, because the type of syntactic dependences they exercise are still restricted. For any sequence of the letters "C" and "D", it is simple to construct an example of finitely demonstrated semantic dependence for which the corresponding strong syntactic dependence is realized by only one chain of direct data flow dependences and direct strong control dependences, whose ith dependence is a direct strong control dependence (direct data flow dependence) if the ith letter of the sequence is "C" (is "D"). This means that, in general, almost arbitrarily complex chains of data flow and control dependences may have to be exercised to reveal operator faults.

As noted at the beginning of Section VI, it follows from conclusions 1 and 3 that syntactic dependences provide a nontrivial bound on the set of statements that can be affected by an operator fault at a given statement. One might think that to remedy the aforementioned weakness of the data flow coverage criteria it is only necessary to extend them to exercise *all* syntactic dependences upon a potentially faulty definition, instead of just data flow dependences. However, the number of tests required to adequately exercise all syntactic dependences can be impractically large, for the following reasons:

- The number of syntactic dependences in a program can be quadratic in the number of statements.
- A given syntactic dependence may be demonstrated by many program walks (even infinitely many), only one of which is associated with erroneous information flow.
- Erroneous information flow via a particular syntactic dependence may depend not only on which walk is executed, but also on the particular input used.

At the very least, the bounds on erroneous information flow that are implied by a program's syntactic dependences *can* be used to filter out test data, selected without regard to the dependences, that cannot possibly reveal certain faults. More ambitiously, it may be possible to *base* test data selection on the determination of syntactic dependences, by analyzing individual syntactic dependences to determine which dependences, which walks demonstrating them, and which inputs executing these walks are most likely to be associated with erroneous information flow. Implementing this approach might require the development of techniques for more precise semantic analysis of programs and for acquiring reliable statistical information about programmer error-making behavior.

2) Detection of Dependence Faults: Dependence analysis has something to say about other kinds of faults besides operator faults. A *dependence fault* is a fault causing different dependences to exist in an incorrect version of a program than in the correct program. A dependence fault may cause either "missing" or "spurious" dependences or both. A dependence is *missing* if it occurs in the correct program but not in the faulty one; it is *spurious* if the reverse is true. For example, if the wrong variable name, say **X**, is used on the left-hand side of an assignment statement, this makes every use of **X** reached by this definition of **X** spuriously data flow dependent on it. This fault may also cause data flow dependences upon definitions killed by the erroneous definition of **X** to be missing from the faulty program, although missing and spurious dependences do not always accompany each other. Dependence faults may also involve control dependence, as when a statement is erroneously placed in the body of a conditional branch statement, causing a spurious control dependence. In a sense, dependence faults are the complement of operator faults. Operator faults change only the semantics of a single statement; they do not change a program's def/use graph. Under certain assumptions, it can be shown that any fault that does change a program's def/use graph changes the program's syntactic dependences as well, and is therefore a dependence fault.

The data flow coverage criteria address dependence faults by exercising potentially spurious data flow dependences. To evaluate the soundness of this approach, it is necessary to relate the semantic and syntactic effects of dependence faults. The results of Section V do this for certain types of dependence faults. If a fault causes the function computed by a statement s to be erroneously relevant to the execution behavior of a statement s', it follows from Theorems 2 and 4 that s' is syntactically dependent on s, since s' is semantically dependent on s. If this syntactic dependence exists only by virtue of the fault, then it is spurious. A fault may also cause the function computed by a statement to be erroneously *irrelevant* to the execution behavior of another statement, with the result that a syntactic dependence between the statements is missing by virtue of the fault. By conclusions 1 and 3, if a statement s is not syntactically dependent on a statement s', then the semantics of s' *is* irrelevant to the execution

behavior of s. Thus, the presence or absence of syntactic dependences may be evidence of erroneous semantic relationships.

Exercising syntactic dependences, as the data flow coverage criteria do, may reveal when the function computed by one statement is erroneously relevant on the execution behavior of another statement, by exercising a spurious syntactic dependence associated with this fault and thereby eliciting the fault's effects. Of course, it is not sure to. Exercising syntactic dependences may also reveal when the function computed by a statement is erroneously irrelevant to the execution behavior of another statement, particularly if, as is often the case, missing syntactic dependences caused by this fault are accompanied by spurious ones. However, because the data flow coverage criteria exercise only restricted types of syntactic dependences, the results of Section V imply that these criteria do not necessarily exercise all missing and spurious syntactic dependences. In the absence of additional information, the only way to remedy this is to exercise every syntactic dependence in a program. For the reasons given in Section VI-A-1, we believe that this approach is untenable. Nevertheless, determination of syntactic dependences might be used to *guide* more discriminating methods for detecting dependence faults, in ways similar to those proposed in Section VI-A-1 for the detection of operator faults.

B. Anomaly Detection

A *program anomaly* is a syntactic pattern that is often evidence of a programming error, irrespective of a program's specification [9]—for example, a variable being used before it has been defined. Korel [15] proposes using program dependences for the detection of "useless" statements, a type of anomaly detection. *Useless statements* are ones that cannot influence the output of a program and can be removed without changing the function the program computes. Korel claims that a statement is useless if there is no output statement strongly syntactically dependent upon it.

Korel did not prove this informal claim. Nevertheless, conclusion 3 supports a version of it: if no output statement in a program is strongly syntactically dependent on s, then the semantics of s is irrelevant to the *values of variables* output by the program. This is because if a change to the semantics of s affected the value of a variable output at statement s', then this would *finitely demonstrate* that s' was semantically dependent on s. A change to the semantics of a statement can affect the output of a program in ways that imply nonfinitely-demonstrated semantic dependence, however. By conclusion 2, nonfinitely-demonstrated semantic dependence might not be accompanied by strong syntactic dependence. In the factorial program of Fig. 2, changing the branch condition of the *while*-loop to **N = N** causes the loop to execute forever; consequently, statement 6 is not executed. Thus vertex v_6 of the program's def/use graph is semantically dependent on vertex v_3. However, this semantic dependence is not finitely demonstrated, and v_6 is not strongly syntactically dependent on v_3. Hence, the fact that an output statement is not strongly syntactically dependent on a statement s does not imply that the semantics of s is irrelevant to the behavior of the output statement. However, if no output statement in a program is *weakly* syntactically dependent on s then, by conclusion 1, the semantics of s *is* irrelevant to the program's output.

C. Debugging and Maintenance

In both software debugging and maintenance, it is often important to know when the semantics of one statement can affect the execution behavior of another statement. In debugging, one attempts to determine what statement(s) caused an observed failure. In maintenance, one wishes to know whether a modification to a program will have unanticipated effects on the program's behavior; to determine this, it is helpful to know what statements are affected by the modified ones and what statements affect the modified ones. There are no general procedures for determining absolute answers to these questions, but dependence analysis can be used to answer them approximately.

In his work on program slicing, Weiser proposes that program dependences be used to determine the set of statements in a program—called a "slice" of the program—that are potentially relevant to the behavior of given statements [26]–[28]. Weiser demonstrates how program slices can be used to locate faults when debugging. He claims that if an incorrect state is observed at a statement s, then only those statements that s is strongly syntactically dependent upon could have caused the incorrect state. He argues that by (automatically) determining those statements and then examining them the debugging process can be facilitated.

While most investigators who proposed uses for program dependences made no attempt to justify these uses rigorously, Weiser [26] did recognize the need to do this for the use of dependences in his program slicing technique, and he attempted to provide such justification via both mathematical proofs and a psychological study. To this end, Weiser implicitly defined a type of semantic dependence and examined its relationship to syntactic dependence. Unfortunately, the mathematical part of Weiser's work is flawed. In his dissertation [26], Weiser states a theorem similar to Theorem 2.[8] In his attempted proof of this theorem, Weiser actually assumes, without proof, that strong syntactic dependence is a necessary condition for semantic dependence. Besides being very close to what Weiser is trying to prove, this assumption is false. Weiser does not address the issue of formal justification for slicing in his subsequent writings.

If a program failure observed at one statement is caused by an operator fault at another statement, it follows from conclusion 1 that the search for the fault can be facilitated

[8]The theorem is stated in terms of Weiser's problematic "color dominance" characterization of control dependence, which he abandoned in his later writings on slicing, in preference to strong control dependence.

by determining weak syntactic dependences, since the statement where the failure was observed is weakly syntactically dependent upon the faculty statement. If the failure implies finitely demonstrated semantic dependence, it follows from conclusion 3 that strong syntactic dependence can be used to help locate the fault. However, if the failure implies a semantic dependence that is not finitely demonstrated, then strong syntactic dependence cannot necessarily be used to locate the fault. This is illustrated by the example in Section VI-B. Hence, for locating operator faults, Weiser's use of strong syntactic dependence in slicing is justified only when the faults cause failures that finitely demonstrate semantic dependence. In his thesis [26], Weiser does not restrict the type of semantic dependence he attempts to localize with slicing to be finitely demonstrated. In [28], however, Weiser defines slicing for terminating programs only. In general, of course, faulty programs may fail to terminate, so this restriction limits the applicability of slicing.

The implications of conclusions 1–4 for maintenance are similar to those for debugging. If a modification involves only the semantics of a single statement, then, by conclusions 1 and 3, only those statements that are syntactically dependent on the statement to be modified could be affected by the modification. Similarly, only those statements that a modified statement is syntactically dependent on could be relevant to the behavior of the modified statement.

VII. Other Related Work

In this section we briefly survey related work, not considered above, on the use of program dependences in testing, debugging, and maintenance.

Bergeretti and Carré [3] present a variant of dependence analysis, called "information flow analysis," that applies to structured programs. They suggest several uses for it, including testing and debugging. They define, by structural induction on the syntax of a programming language, three information flow relations that are similar to strong syntactic dependence.

Recently, several papers have investigated the semantic basis for proposed uses of program dependences [4], [13], [14], [23], [25]. Some of these papers address the use of dependences in software debugging and maintenance. Horwitz *et al.* [13] present a theorem that characterizes when two programs with the same dependences compute the same function. Reps and Yang [23] use a version of this result to prove two theorems about program slicing. One of these states that a slice of a program computes the same function as the program itself on inputs for which the computations of both the program and its slice terminate. The second theorem states that if a program is decomposed into slices, the program halts on any input for which all of the slices halt. The latter two theorems are used by Horwitz *et al.* [14] to justify an algorithm for integrating versions of a program.

This paper differs in three respects from the other recent work on the semantic basis for the use of dependences. First, the other work does not address the concept of semantic dependence. Second, while the results in those papers are proved for a simple, structured programming language, we adopt a graph-theoretic framework for our results, similar to that in [26], that makes them applicable to programs of any procedural programming language and to unstructured programs as well as structured ones. Third, this paper considers the semantic significance of both weak and strong control dependence, while the above papers consider only strong control dependence.

VIII. Conclusion

In summary, we have presented several results clarifying the significance of program dependences for the execution behavior of programs. We have shown that two generalizations of both control and data flow dependence, called weak and strong syntactic dependence, are necessary conditions for certain interstatement relationships involving the effects of program faults and modifications. This implies that weak and strong syntactic dependences, which can be computed efficiently, may be used to guide such activities as test data selection and program debugging. On the other hand, we have also shown that neither data flow nor syntactic dependence is a sufficient condition for the interstatement relationships in question. This result discourages the use of such dependences, in the absence of additional information, as evidence for the presence of these relationships. Finally, we have shown that strong syntactic dependence is not a necessary condition for some interstatement relationships involving program nontermination, and this suggests that some proposed uses of strong syntactic dependence in debugging and anomaly detection are unjustified.

There are several possible lines of further investigation related to the use of program dependences in testing, debugging, and maintenance. For example, our results could be usefully extended to provide information about the effects of larger classes of faults and program modifications. Another possible line of investigation is the development of testing methods that exploit the fact that syntactic dependence bounds the statements that are affected by certain type of faults. For example, Morell [18], Richardson and Thompson [24], and Demillo *et al.* [6] propose test data selection methods that might be adapted to do this, since their methods are based on determining conditions for erroneous program states to occur and then propagate to a program's output. A third possible line of investigation is the development of more sophisticated semantic analysis techniques to complement dependence analysis.

Appendix

Sketch of the Proofs of Theorems 2 and 4

The proofs of Theorems 2 and 4 are lengthy, so we only sketch them here; complete proofs are found in [21].

To describe the proofs of Theorems 2 and 4, it is necessary to present the complete definition of semantic dependence. This definition uses some notation that we now

informally define. An interpretation I of a def/use graph $G = (G, \Sigma, D, U)$ is triple $(\mathfrak{D}, F, N)$. $\mathfrak{D}$ is the set of objects which serve as the inputs, outputs, and intermediate results of a computation, and is called the *domain* of I. F is function that associates with every vertex $v \in V(G)$ a partial recursive function $F(v)$ that maps an assignment of values for the variables in $U(v)$ to an assignment of values for the variables in $D(v)$. F represents the ability of a program statement to alter the value of variables. N is a function that associates with every decision vertex $d \in V_{dec}(G)$ a function $N(d)$ mapping an assignment of values for the variables in $U(d)$ to a successor of d. N represents the ability of a branch condition of a conditional branch statement to determine the order of statement execution. The pair $P = (G, I)$ is called a *program*. The computation sequence induced by executing P on an input $d \in \mathfrak{D}^{\Sigma}$ is denoted $\mathcal{C}_P(d) = \{(v_i, val_i)\}$.[9] The execution history of $v \in V(G)$ induced by $\mathcal{C}_P(d)$ is denoted $\mathcal{H}_P(v, d)$, and its ith element, if it exists, is denoted $\mathcal{H}_P(v, d)(i)$.

We are now ready to present the formal definition of semantic dependence.

Definition 12: Let $G = (G, \Sigma, D, U)$ be a def/use graph, and let $u, v \in V(G)$. Vertex u is *semantically dependent* on vertex v iff there exist interpretations $I_1 = (\mathfrak{D}, F_1, N_1)$ and $I_2 = (\mathfrak{D}, F_2, N_2)$ of G and an input $d \in \mathfrak{D}^{\Sigma}$ such that, letting $P_1 = (G, I_1)$ and $P_2 = (G, I_2)$, both of the following conditions are satisfied:

1) For all $w \in V(G) - \{v\}$, $F_1(w) = F_2(w)$ and if $w \in V_{dec}(G)$ then $N_1(w) = N_2(w)$.
2) Either of the following conditions is satisfied:
   a) There is some $i \geq 1$ such that $\mathcal{H}_{P_1}(u, d)(i)$ and $\mathcal{H}_{P_2}(u, d)(i)$ are both defined but are unequal.
   b) $\mathcal{H}_{P_1}(u, d)$ is longer than $\mathcal{H}_{P_2}(u, d)$, and either $\mathcal{C}_{P_2}(d) = \{(v_i, val_i)\}$ is infinite or it visits some vertex v_i from which u is unreachable.

I_1, I_2, and d are said to *demonstrate* that u is semantically dependent on v. If condition 1 holds and either condition 2(a) holds or both of the following conditions are true:

1) $\mathcal{H}_{P_1}(u, d)$ is longer than $\mathcal{H}_{P_2}(u, d)$,
2) u is unreachable from some vertex of $\mathcal{C}_{P_2}(d)$,

then I_1, I_2, and d are said to *finitely demonstrate* that u is semantically dependent on v.

The last part of condition 2(b) may be intuitive. It requires that if a semantic dependence is demonstrated by the fact that one execution history is longer than another, then the computation sequence corresponding to the shorter history must be infinite or must contain a vertex from which the dependent vertex is unreachable. This requirement prevents a semantic dependence from being demonstrated solely as the result of the divergence of a function assigned to a vertex. Such divergence can cause an execution history to be shorter than it would be if the divergence did not occur, by terminating a computation. The reason this possibility is disallowed is that if it were not, then each vertex would be semantically dependent on every other vertex from which it is reachable via an acyclic initial walk (path), trivializing the semantic dependence relation. If the shorter of two execution histories demonstrating a semantic dependence corresponds to either an infinite computation or to a computation containing a vertex from which the dependent vertex is unreachable, then divergence at a vertex either does not occur or is irrelevant to the demonstration of the dependence, respectively.

The proofs of Theorems 2 and 4 have the following basic form. First, a graph-theoretic structure is defined which represents the potential flow of data to a vertex v along an initial walk Wv in a def/use graph G. This structure is called the "context" of v with respect to Wv. It is then shown that in any execution of a program with def/use graph G, the arguments to an execution of v [that is, the values of the variables in $U(v)$] are completely determined by its context. Next, necessary conditions for semantic dependence are given in terms of walks and contexts. These are obtained by analyzing the pair of possibly infinite "walks" executed by a pair of interpretations and an input that demonstrate semantic dependence. Finally, these conditions are used to prove Theorems 2 and 4. We now describe each of these steps in more detail.

The *context* $CON(v, Wv)$ of a vertex v with respect to an initial walk Wv in a def/use graph $G = (G, \Sigma, D, U)$ is a directed tree (technically an "in-tree" [12]) that represents the cumulative flow of data to v along W. $CON(v, Wv)$ contains a distinguished vertex of outdegree zero, called its *sink*. Each vertex of $CON(v, Wv)$ is labeled with a vertex of G and each arc of $CON(v, Wv)$ is labeled with a variable of Σ. $CON(v, Wv)$ is defined inductively as follows. If W is empty then $CON(v, Wv)$ consists of a single vertex labeled "v". Otherwise, $CON(v, Wv)$ consists of

1) A sink s labeled "v".
2) For each variable $\sigma \in U(v)$ such that $W = XuY$ with $\sigma \in D(u) - D(Y)$, a copy of $CON(u, Xu)$ and an arc from its sink to s labeled "σ".

To see the significance of a context $CON(v, Wv)$, notice that if G represents a "real" program P then $CON(v, Wv)$ is analogous to the set of symbolic values held by the variables of $U(v)$ after the instruction sequence $I(W)$ of P corresponding to W is *symbolically executed* [5] (equivalently, executed under a Herbrand interpretation [11], [17]). These symbolic values define the actual values of the variables in $U(v)$, as functions of the inputs to P, when $I(W)$ is executed normally; hence, the symbolic values determine the actual ones. In the same way, the interpretation of the vertex labels of the context $CON(v, Wv)$ determines the values of the variables in $U(v)$ when Wv is executed for a given input and (abstract) interpretation of G. This is demonstrated formally by induction on the length of W.

To state necessary conditions for semantic dependence in terms of walks and contexts, it is necessary to introduce three auxiliary concepts: "hyperwalks," "consistency," and "reciprocal consistency." Since program computa-

[9] The symbol $\mathfrak{D}^{\Sigma}$ denotes the set of functions from Σ into $\mathfrak{D}$.

tions may fail to terminate, it is necessary to consider infinite "walks" in def/use graphs; a *hyperwalk* in a def/use graph G is sequence of vertices that is either an ordinary walk or an infinite one. A hyperwalk W is *consistent* if there are no two occurrences of a decision vertex d in W that have the same context but are followed by different successors of d. This notion is analogous to the notion of path consistency in program schema theory [11]. Because the context $CON(d, Xd)$ determines the values of the variables in $U(d)$ when Xd is executed, and hence determines the branch taken at d, an executable hyperwalk must be consistent.[10] Reciprocal consistency is similar to consistency, but is a necessary condition for a pair of hyperwalks to be executed by a pair of interpretations I_1 and I_2 that differ at only one vertex—such as a pair of interpretations that demonstrate semantic dependence. A pair of hyperwalks W and X is *reciprocally v-consistent* if

$$W = W_1 u u' W_2 \quad \text{and} \quad X = X_1 u u'' X_2 \quad \text{and} \quad u' \neq u''$$

implies that either $CON(u, W_1 u) \neq CON(u, X_1 u)$ or that $CON(u, W_1 u)$ contains a vertex labeled "v". Intuitively, if W and X are executed by I_1 and I_2, respectively, the only way that $CON(u, W_1 u) = CON(u, X_1 u)$ can hold is if v is the vertex whose interpretation differs between I_1 and I_2 and data flows from v to u via $CON(u, W_1 u)$ and $CON(u, X_1 u)$.

Having defined contexts, hyperwalks, consistency, and reciprocal consistency, and having established their relevance to program execution, it is possible to establish necessary conditions for semantic dependence in terms of walks and contexts.

Theorem 6: Let $G = (G, \Sigma, D, U)$ be a def/use graph and $u, v \in V(G)$. Then u is semantically dependent on v iff there exist initial hyperwalks W and X in G such that each of the following is true:

1) W and X are algorithmically listable; and
2) W and X are consistent and reciprocally v-consistent; and
3) At least one of the following conditions holds:
   a) $W = W_1 u W_2$ and $X = X_1 u X_2$, where u occurs the same number of times in $W_1 u$ as in $X_1 u$, and where either $CON(u, W_1 u) \neq CON(u, X_1 u)$ or $CON(u, W_1 u)$ contains a vertex other than its sink that is labeled "v".
   b) W contains more occurrences of u than X does, and either X is infinite or X contains a vertex from which u is unreachable.

It is interesting to note that these conditions are also *sufficient* for semantic dependence [21], although this fact is not used in proving the results stated in Section V. Condition 1 of Theorem 6, which means that there are some (possibly nonterminating) algorithms for listing W and X, is required for proving the sufficiency of the conditions but not for proving the results of Section V; hence, we do not discuss it further.

[10]It can also be shown that every consistent hyperwalk is executed by some interpretation, using what is essentially a Herbrand interpretation of a def/use graph.

To prove that the conditions of Theorem 6 are necessary for semantic dependence, one lets the hyperwalks W and X of the theorem be the walks executed by interpretations I_1 and I_2 of G, respectively, that, in conjunction with some input d, demonstrate that u is semantically dependent on v. Since these walks are executed by I_1, I_2, and d, they must be consistent and reciprocally v-consistent, for the reasons given above; hence W and X satisfy condition 2 of Theorem 6. Note that condition 3 of Theorem 6 mirrors condition 2 of the definition of semantic dependence; however, the former condition is syntactic, whereas the latter is semantic. Let $P_1 = (G, I_1)$ and $P_2 = (G, I_2)$.

Suppose that condition 2(a) of the definition of semantic dependence is satisfied:

> There is some $i \geq 1$ such that $\mathcal{HC}_{P_1}(u, d)(i)$ and $\mathcal{HC}_{P_2}(u, d)(i)$ are defined and unequal.

Let $W = W_1 u W_2$ and $X = X_1 u X_2$, where u occurs exactly i times in $W_1 u$ and $X_1 u$. Suppose v is not a vertex label in $CON(u, W_1 u)$. Intuitively, then, data does not flow along W_1 from v to the last occurrence of u in $W_1 u$. Since v is the only vertex whose interpretation changes between I_1 and I_2, and since the interpretation of the vertex labels of $CON(u, W_1 u)$ and $CON(u, X_1 u)$ determines the values of the variables in $U(u)$ when $W_1 u$ and $X_1 u$, respectively, are executed for a particular input, $CON(u, W_1 u) \neq CON(u, X_1 u)$. Otherwise, we would have $\mathcal{HC}_{P_1}(u, d)(i) = \mathcal{HC}_{P_2}(u, d)(i)$. If v is a vertex label in $CON(u, W_1 u)$, then it is possible that $CON(u, W_1 u) = CON(u, X_1 u)$, since different data could flow from v to u via this context under I_1 than under I_2. Thus, if condition 2(a) of the definition of semantic dependence is satisfied then condition 3(a) of Theorem 6 is also satisfied.

On the other hand, suppose that condition 2(b) of the definition of semantic dependence is satisfied:

> $\mathcal{HC}_{P_1}(u, d)$ is longer than $\mathcal{HC}_{P_2}(u, d)$, and either $\mathcal{C}_{P_2}(d)$ is infinite or it visits a vertex from which u is unreachable.

Then clearly condition 3(b) of Theorem 6 is satisfied.

Having established Theorem 6, we show that various subconditions of condition 3 of the theorem, taken together with condition 2, imply various types of syntactic dependence. We assume henceforth that condition 2 is satisfied by W and X.

It is easy to see that if condition 3(a) of Theorem 6 is satisfied by W and X because $CON(u, W_1 u)$ contains a vertex other than its sink that is labeled "v", then u is data flow dependent on v. This is because the head of an arc of $CON(u, W_1 u)$ is directly data flow dependent on its tail, as is clear from the definition of a context. The other subcondition of 3(a), $CON(u, W_1 u) \neq CON(u, X_1 u)$, implies that u is strongly syntactically dependent on v, but this is more difficult to see. The proof of this

fact is a pivotal element in establishing the results of Section V, since a generalization of condition 3(a) arises in considering condition 3(b).

Let us refer to the subcondition $CON(u, W_1u) \neq CON(u, X_1u)$ of condition 3(a) in Theorem 6 as subcondition 3(a)'. The first step in showing that 3(a)' implies that u is strongly syntactically dependent on v is to show that 3(a)' implies that some vertex-label of $CON(u, W_1u)$ or $CON(u, X_1u)$ is strongly control dependent on a decision vertex d having kth occurrences in W_1u and X_1u that are followed by different successors. Intuitively, d makes branches that cause $CON(u, W_1u)$ and $CON(u, X_1u)$ to differ. Since u is data flow dependent on the vertex labels of its contexts, u is strongly syntactically dependent on d. The existence of d is established by Lemma 1 below. Since W and X are reciprocally v-consistent, it follows either that d is data flow dependent on v or that the initial walks $W_{1,1}d$ and $X_{1,1}d$ of the lemma, which are shorter than W_1u and X_1u, themselves satisfy the hypothesis of the lemma. Since strong syntactic dependence is a transitive relation, it is evident that an inductive proof that u is strongly syntactically dependent on v can be framed using the lemma. The formalization of this proof is relatively straightforward; hence we focus on the lemma and its proof.

Lemma 1: Let $G = (G, \Sigma, D, U)$ be a def/use graph with $u \in V(G)$, and let W_1u and X_1u be initial walks in G containing the same number of occurrences of u. If $CON(u, W_1u) \neq CON(u, X_1u)$, then there is a vertex $d \in V(G)$ such that a vertex-label of $CON(u, W_1u)$ or $CON(u, X_1u)$ is strongly control dependent on d and such that $W_1u = W_{1,1}dd'W_{1,2}$ and $X_1u = X_{1,1}dd''X_{1,2}$, where $d' \neq d''$ and d occurs equally often in $W_{1,1}d$ and $X_{1,1}d$.

This lemma is proved by assuming that no such vertex d exists, and then showing that this implies that W_1u and X_1u have a special structure which precludes u having different contexts with respect to them, which would of course be a contradiction. More precisely, we show, by induction on the length of the longer of W_1u and X_1u, that if there is no such vertex d then there are walks $R_0, R_1, \cdots, R_n, S_1, S_2, \cdots, S_n$, and $T_1, T_2, \cdots, T_n$ satisfying each of the following conditions:

1) $W_1u = R_0S_1R_1S_2R_2 \cdots S_nR_n$.

2) $X_1u = R_0T_1R_1T_2R_2 \cdots T_nR_n$.

3) For $i = 1, 2, \cdots, n$, R_i begins with $ifd(r_{i-1})$, where r_{i-1} is the last vertex of R_{i-1}, and $ifd(r_{i-1})$ does not occur in S_i or T_i.

4) For $i = 1, 2, \cdots, n$, the first vertex of S_iR_i is different from the first vertex of T_iR_i.

It follows that for $i = 1, 2, \cdots, n$, S_i and T_i consist of vertices that are strongly control dependent on r_{i-1}. It is not difficult to show, using induction and the transitivity of strong control dependence, that if a vertex label of $CON(u, W_1u)$ or $CON(u, X_1u)$ occurred in S_i or T_i, then we could let d in the statement of Lemma 1 be some r_j, where $j < i$, to obtain a contradiction. Intuitively, this means that the only part of W_1u that is relevant to the structure of $CON(u, W_1u)$, namely the subsequence $R_0, R_1, \cdots, R_n$, is identical to the only part of X_1u that is relevant to the structure of $CON(u, X_1u)$. This implies that the two contexts are identical—which we show formally by induction on the length of W_1u, exploiting the inductive definition of a context. Since this contradicts the hypothesis of the lemma, we conclude the lemma is true. This concludes our sketch of the proof that condition 3(a) of Theorem 6 implies that u is strongly syntactically dependent on v.

Suppose that condition 3(b) of Theorem 6 is satisfied by W and X. This condition is the disjunction of two subconditions, which we will denote 3(b)' and 3(b)''. Subcondition 3(b)' is

> W contains more occurrences of u than X does, and X is infinite

while subcondition 3(b)'' is

> W contains more occurrences of u than X does, and X contains a vertex from which u is unreachable.

We show that 3(b)' implies, in conjunction with condition 2 of Theorem 6, that u is weakly syntactically dependent on v (it may or may not be strongly syntactically dependent on v) and that 3(b)'' implies that u is strongly syntactically dependent on v. We now sketch these proofs, beginning with that of the second result.

Suppose that subcondition 3(b)'' is satisfied. To deal with this case, we prove Lemma 2 below. Note that if we identify the vertex u of 3(b)'' with the vertex w of the lemma and let the walks Yw and Zx of the lemma be appropriate prefixes of W and X, respectively, then the lemma applies. The reciprocal v-consistency of W and X implies that either the vertex d of the lemma is data flow dependent on v, which implies that u is strongly syntactically dependent on v, or the walks Y_1d and Z_1d of the lemma satisfy the hypothesis of Lemma 1. Since Y_1d and Z_1d are shorter than Yw and Zx, respectively, this allows us to frame an inductive proof that u is strongly syntactically dependent on v, similar to that discussed with regard to subcondition 3(a)' of Theorem 6.

Lemma 2: Let G be a control flow graph, $w, x \in V(G)$, and Yw and Zx walks in G. If 1) w is unreachable from x, 2) Yw and Zx begin with the same vertex, and 3) w has more occurrences in Yw than in Zx, then there is a vertex d such that a) w is strongly control dependent on d, b) $Yw = Y_1dd'Y_2$, and c) $Zx = Z_1dd''Z_2$, where $d' \neq d''$ and d has the number of occurrences in Y_1d as in Z_1d.

The proof of this lemma implicitly demonstrates that if no such vertex d exists, then Yw and Zw have a special structure similar to that discussed above in regard to the proof of Lemma 1, although the proof of Lemma 2 proceeds directly instead of by contradiction. It is shown that if R is the longest common prefix of Yw and Zx then $Yw = RY'w$ and $Zx = RZ'x$, where the first vertex of $Y'w$ is different from that of $Z'x$. If w is strongly control dependent on the last vertex r of R then we may let $d = r$.

Suppose that w is not strongly control dependent on r. It is shown that in the case $ifd(r)$, the immediate forward dominator of r, occurs in both $Y'w$ and $Z'x$. Thus, $Y'w = Y_1 ifd(r) Y_2$ and $Z'x = Z_1 ifd(r) Z_2$, where $ifd(r)$ does not occur in Y_1 or Z_1. Each vertex in Y_1 and Z_1 is strongly control dependent on r, so w cannot occur in either walk. This implies that there are more occurrences of w in $ifd(r) Y_2$ than in $ifd(r) Z_2$. Since these walks satisfy the hypothesis of Lemma 2 and are shorter than Yw and Zx, respectively, we may frame an inductive proof of the lemma. By assuming the truth of the lemma for $ifd(r) Y_2$ and $ifd(r) Z_2$, we conclude that $ifd(r) Y_2 = S_1 dd' S_2$ and $ifd(r) Z_2 = T_1 dd'' T_2$, where $d \neq d''$ and d has the same number of occurrences in $S_1 d$ as in $T_1 d$. The vertex d must also have the same number of occurrences $RY_1 S_1 d$ as in $RZ_1 T_1 d$—for otherwise d occurs in Y_1 or Z_1 and so is strongly control dependent on r, which, by the transitivity of strong control dependence, implies that w is strongly control dependent on r.

To demonstrate the implications of subcondition 3(b)′ of Theorem 6 it is necessary to introduce a new type of control dependence, called "exit dependence." The exit dependence relation represents the potential ability of a loop exit condition to determine whether a statement outside the loop is executed, by determining whether the loop terminates.

Definition 13: Let G be a control flow graph, and let $u, v \in V(G)$. Vertex u is *exit dependent* on vertex v iff there is a cycle C and a walk vWu in G such that v occurs in C and such that Wu is vertex-disjoint from C.

For example, in the def/use graph of Fig. 2, vertex v_7 is exit dependent on vertex v_3, as can be seen by letting $u = v_7$, $v = v_3$, $C = v_3 v_4 v_5 v_3$, and $vWu = v_3 v_6 v_7$.

In [21], it is shown that the weak control dependence relation of a control flow graph G is the transitive closure of the union of the exit dependence and strong control dependence relations of G. That is, the existence of a chain of exit dependences and strong control dependences from u to v indicates u is weakly control dependent on v, and conversely if u is weakly control dependent on v then such a chain exists. We very briefly describe the basis for this result. A preliminary step in establishing the result is showing that the strong control dependence relation is the transitive closure of the "direct strong control dependence" relation.

Definition 14: Let G be a control flow graph, and let $u, v \in V(G)$. Vertex u is *directly strongly control dependent* on vertex v iff v has successors v' and v'' such that u forward dominates v' but u does not forward dominate v''.

The similarity between this definition and that of direct weak control dependence is obvious, as is the similarly between the definition of weak control dependence and the characterization of strong control dependence in terms of direct strong control dependence. The difference between direct weak and direct strong control dependence, and therefore between weak and strong control dependence, is that u can be directly weakly control dependent on v because there are infinite walks not containing u that begin with one successor of v but no such walks that begin with the other successor. It can be shown that any such infinite walk contains a cycle that demonstrates that u is exit dependent on either v or some vertex strongly control dependent on v.

Suppose now that subcondition 3(b)′ of Theorem 6 is satisfied by W and X, along with condition 2 of that theorem. The following lemma, which is proved by an argument similar to those used to establish Lemmas 1 and 2, shows that subcondition 3(b)′ reduces to condition 3(a) of Theorem 6.

Lemma 3: Let G be a control flow graph with $w \in V(G)$, and let Yw and Z be hyperwalks in G such that Yw and Z begin with the same vertex, w has more occurrences in Yw than in Z, Z is infinite, and w is reachable from every vertex in Z. Then there is a vertex $d \in V(G)$ such that each of the following is true:

1) Either w is strongly control dependent on d, w is exit dependent on d, or there is a vertex $x \in V(G)$ such that x is strongly control dependent on d and such that w is exit dependent on x.

2) $Yw = Y_1 dd' Y_2$ and $Z = Z_1 dd'' Z_2$, where $d' \neq d''$ and d has the same number of occurrences in $Y_1 d$ as in $Z_1 d$.

Because the weak syntactic dependence relation for G is the transitive closure of the union of the exit dependence and strong control dependence relations for G, the vertex w of the lemma is weakly control dependent on the vertex d whose existence the lemma asserts. Suppose that w is identified with the vertex u of subcondition 3(b)′, and that the lemma is applied to $Z = X$ and a prefix of Yw of W that contains more occurrences of u than X does. Then either the vertex d whose existence the lemma asserts is data flow dependent on v and therefore u is weakly syntactically dependent on v, or the walks $Y_1 d$ and $Z_1 d$ satisfy the hypothesis of Lemma 1. Thus, using the fact that strong syntactic dependence is transitive and implies weak syntactic dependence, we can frame an inductive proof that u is weakly syntactically dependent on v.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[3] J. F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of **while**-programs," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 37-61, Jan. 1985.

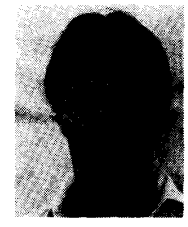
[4] R. Cartwright and M. Felleisen, "The semantics of program dependence," in *Proc. SIGPLAN '89 Conf. Programming Language Design and Implementation*, ACM, New York, 1989, pp. 13-27.

[5] L. A. Clarke and D. J. Richardson, "Symbolic evaluation methods—Implementations and applications," in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, Eds. Amsterdam, The Netherlands: North-Holland, 1981, pp. 65-102.

[6] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, "An extended overview of the Mothra software testing environment," in *Proc. Second Workshop Software Testing, Verification and Analysis*, Banff, Alberta, July 1988, pp. 142-151.

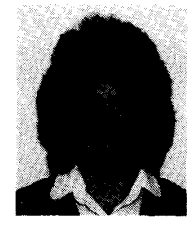
[7] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504-513, July 1977.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 5, pp. 319-349, July 1987.
[9] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *ACM Comput. Surveys*, vol. 8, no. 3, pp. 306-330, Sept. 1976.
[10] P. G. Frankl, "The use of data flow information for the selection and evaluation of software test data," Doctoral dissertation, New York Univ., New York, 1987.
[11] S. A. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification*. Berlin: Springer-Verlag, 1975.
[12] F. Harary, *Graph Theory*. Reading, MA: Addison-Wesley, 1969.
[13] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," in *Proc. Fifteenth ACM Symp. Principles of Programming Languages*, ACM, New York, 1988, pp. 146-157.
[14] ——, "Integrating non-interfering versions of programs," in *Proc. Fifteenth ACM Symp. Principles of Programming Languages*, ACM, New York, 1988, pp. 133-145.
[15] B. Korel, "The program dependence graph in static program testing," *Inform. Processing Lett.*, vol. 24, pp. 103-108, Jan. 1987.
[16] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. SE-9, no. 3, pp. 347-354, May 1983.
[17] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.
[18] L. J. Morell, "A theory of error-based testing," Doctoral dissertation, Univ. Maryland, College Park, 1984.
[19] S. C. Ntafos, "On required element testing," *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 795-803, Nov. 1984.
[20] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, no. 12, pp. 1184-1201, Dec. 1986.
[21] A. Podgurski, "The significance of program dependences for software testing, debugging, and maintenance," Doctoral dissertation, Dep. Comput. Inform. Sci., Univ. Massachusetts, Amherst, 1989.
[22] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11, no. 4, pp. 367-375, Apr. 1985.
[23] T. Reps and W. Yang, "The semantics of program slicing," Univ. of Wisconsin-Madison, Tech. Rep., 1989.
[24] D. J. Richardson and M. C. Thompson, "The RELAY model of error detection and its application," in *Proc. Second Workshop Software Testing, Verification, and Analysis*, IEEE Computer Society, Los Angeles, CA, 1988.
[25] R. P. Selke, "A rewriting semantics for program dependence graphs," in *Conf. Rec. 16th ACM Symp. Principles of Programming Languages*, ACM, New York, 1989, pp. 12-24.
[26] M. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Doctoral dissertation, Univ. Michigan, Ann Arbor, 1979.
[27] ——, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446-452, July 1982.
[28] ——, "Program slicing," *IEEE Trans. Software Eng.*, vol. SE-10, no. 4, pp. 352-356, July 1984.



Andy Podgurski received the M.S. and Ph.D. degrees in computer science from the University of Massachusetts at Amherst in 1985 and 1989, respectively.

He is currently an Assistant Professor at the Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH. His research interests include software engineering, software validation, programming languages and translators, and the automated semantic analysis of programs.



Lori A. Clarke received the B.A. degree in mathematics from the University of Rochester, Rochester, NY, and the Ph.D. degree in computer science from the University of Colorado, Boulder.

She worked as a programmer for the University of Rochester, School of Medicine, and for the National Center for Atmospheric Research. Since 1975 she has been on the faculty in the Department of Computer and Information Science at the University of Massachusetts, where she currently holds the rank of professor. She is Director of the University's Software Development Laboratory, which is exploring a range of software engineering issues. Her primary research areas are software testing and validation and software development environments.

Dr. Clarke is a former IEEE Distinguished Visitor and ACM National Lecturer. Currently she is Vice-Chair of SIGSOFT, a member of the IEEE Technical Committee on Software Engineering, and an Associate Editor of *TOPLAS*.