The Implications of Program Dependences for Software Testing, Debugging, and Maintenance

Andy Podgurski[†]
Lori A. Clarke[‡]

[†]Computer Engineering & Science Department
Case Western Reserve University
Cleveland, Ohio 44106

[‡]Software Development Laboratory
Computer & Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

This paper presents a formal, general model of program dependences. Two generalizations of control and data dependence, called weak and strong syntactic dependence, are presented. Some of the practical implications of program dependences are determined by relating weak and strong syntactic dependence to a relation called semantic dependence. Informally, one program statement is semantically dependent on another if the latter statement can affect the execution behavior of the former. It is shown that weak syntactic dependence is a necessary condition for semantic dependence, but that neither weak nor strong syntactic dependence are sufficient conditions. The implications of these results for software testing, debugging, and maintenance are then explored.

1 Introduction

Program dependences are syntactic relationships between the statements of a program that represent aspects of the program's control flow and data flow. They are used in several areas of computer science to obtain "approximate" information about semantic relationships between statements. Typically, proposed uses of program dependences have only been justified informally, if at all. Since program dependences are used for such critical purposes as software testing [Lask83, Kore87, Ntaf84, Rapp85], debugging [Berg85, Weis84], and maintenance [Reps89, Weis84], code optimization and parallelization [Ferr87, Padu86], and computer security [Denn77],[1] this lack of rigor is unacceptable. This paper presents a formal, general model of program dependences and investigates its implications for software testing, debugging, and maintenance. These implications validate certain proposed uses of program dependences, invalidate others, and suggest new ones.

For example, one application of program dependence to software testing is the detection of operator faults. An **operator fault** is a fault that affects the operators[2] used by a single program statement, but does not affect the control structure or use of variables in a program. 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. Operator faults are common in programming, and it would be useful to be able to automati-

[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.

cally detect and locate them. Unfortunately, as with many other semantic properties of programs, deciding whether a program contains an operator fault 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. Those necessary conditions are expressed in terms of program dependences.

This paper addresses procedural programs, for which there are two basic types of program dependences: "control dependences", which are properties of a program's control structure, and "data dependences", which are properties of a program's use of variables. **Dependence analysis**, the process of determining a program's dependences, involves control flow analysis and data flow analysis, and can be implemented efficiently.

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', affects the execution behavior of s. The significance of semantic dependence is that it is a necessary condition for certain interstatement semantic relationships. For example, if s and s' are distinct statements, 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.

In Section 2 some necessary terminology is defined and Section 3 defines control, data, and syntactic dependence. Related work is surveyed in Section 4. In Section 5 semantic dependence is defined and then related to syntactic dependence in Section 6. In Section 7 the implications of these results for software testing, debugging, and maintenance are described. Finally, Section 8 presents possible future research directions.

2 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 $V(G) \times V(G) - \{(v,v) | v \in V(G)\}$. 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 **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 $i = 1, 2, \ldots, n-1$. 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 $W = w_1 w_2 \cdots w_n$ and $X = x_1 x_2 \cdots x_n$ are walks such that either W is empty, X is empty, or w_n is adjacent to x_1, then the **concatenation** of W and X, denoted WX, is the walk $w_1 w_2 \cdots w_n x_1 x_2 \cdots x_n$.

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**. If u is a decision vertex and u' is a successor of u, the other successor of u is called the **complement of u' with respect to u**, and is denoted $c_u(u')$.

The vertices of a control flow graph represent a program's statements, and the arcs represent possible transfers of control between the statements. The program's entry and exit points are represented by the initial and final vertices, respectively. A decision vertex

[3] This restriction is made for simplicity only.

```
1.  input (X, Y);
2.  if X > Y then
3.     Max := X;
    else
4.     Max := Y;
    endif;
5.  output (Max);
```

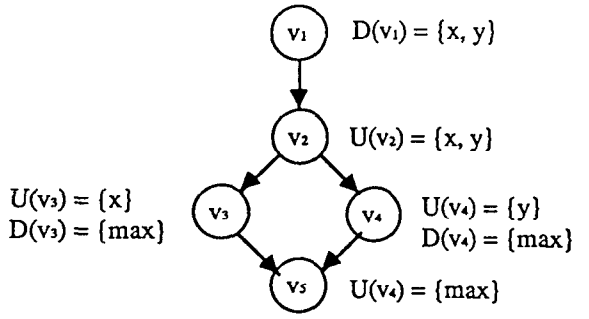

Figure 1: Max program and its def/use graph

represents the branch condition of a conditional branch statement. The definition of a control flow graph given here allows procedural programs of a very general nature to be represented (possibly by including "dummy" vertices that do not correspond to statements).

The control flow graph of the program in Figure 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.*

While the concept of forward dominance has appeared before in the literature, the concept of "strong forward dominance" is apparently new. In the control flow graph of Figure 1, v_5 (strongly) forward dominates each vertex, but v_3 does not forward dominate v_1. In the control flow graph of Figure 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.

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.*

Informally, the "immediate forward dominator" of a decision vertex d is the vertex where all walks leaving d first come together again.

Definition 4 *Let G be a control flow graph. The* **immediate forward dominator** *of a vertex $v \in (V(G) - \{v_F\})$ 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 Figure 1, v_5 is the immediate forward dominator of v_2, v_3, and v_4. In the control flow graph of Figure 2, v_6 is the immediate forward dominator of v_3.

Data, syntactic, and semantic dependence are defined in terms of a "def/use graph", which represents both the control flow in a program and how the program's variables are defined (assigned values) and used (have their values referenced).

Definition 5 *A* **def/use graph** *is a quadruple $\mathbf{G} = (G, \Sigma, D, U)$, where G is a control flow graph, Σ is a finite set of symbols called* **variables**, *and $D: V(G) \rightarrow 2^{\Sigma}$, $U: V(G) \rightarrow 2^{\Sigma}$ are functions.*[4]

For each vertex $v \in V(G)$, $D(v)$ represents the set of variables defined at the statement represented by v, and $U(v)$ represents the set of variables used at that statement. A def/use graph is similar to a program schema [Grei75, Mann74] and is essentially the way a program is represented for data flow analysis [Aho86].

The def/use graph of the program in Figure 1 is shown alongside the program.

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

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

Finally, in discussing dependence relations of various kinds, if an object a is dependent on an object b, we say that b is the **parent** of the dependence and that a is the **child** of the dependence.

[4] We denote the power set (set of all subsets) of a set S by 2^S.

```
1.  input (N);
2.  Fact := 1;
3.  while not N = 0 loop
4.     Fact := Fact * N;
5.     N := N - 1;
    end loop;
6.  output ("The factorial is ");
7.  output (Fact);
```

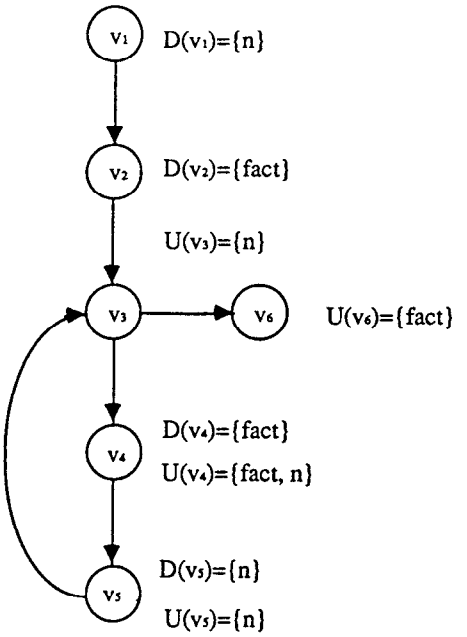

Figure 2: Factorial program and its def/use graph

3 Control, Data, and Syntactic Dependence

3.1 Control Dependence

The concept of control dependence was introduced to model the effect of conditional branch statements on the behavior of programs. Intuitively, a statement s in a program P is control dependent on a statement s' in P if s' is a conditional branch statement and the control structure of P potentially allows s' to decide whether s is executed or not. For example, in the program of Figure 1, statements 3 and 4 are control dependent on statement 2. 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 for a particular 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 and is generally applicable to procedural programming languages. Two definitions that satisfy these requirements are "weak control dependence" and "strong control dependence".

Strong control dependence was originally defined in the context of computer security [Denn77][5], and this definition has been used by several authors [Kore87, Padu86, Weis84]. It was the first graph-theoretic, language and structure-independent characterization of control dependence appearing 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 Figure 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 Figure 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 strong control dependence characterizes: (1) the statements that constitute the "body" of a structured or unstructured conditional branch statement and (2) the branch condition of a loop.

Weak control dependence [Podg89] 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**

[5] In [Denn77] the concept is called "implicit information flow".

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 $v_1, v_2, \ldots, v_n$ 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 $i = 1, 2, \ldots, n-1$.*

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 statement 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 Figure 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. Since an exit statement of a loop potentially determines whether execution of the loop terminates, the additional dependences of the weak control dependence relation are relevant to program behavior. Moreover, there are efficient algorithms for computing both the weak and strong control dependence relations for a control flow graph [Podg89].

3.2 Data Dependence

Although several types of data dependence are discussed in the literature, we consider only "data flow dependence" [Ferr87, Padu86]. Informally, a statement s is data flow dependent on a statement s' if there is a sequence of variable assignments that potentially propagates data from s' to s.

Definition 9 *Let $\mathbf{G} = (G, \Sigma, D, U)$ 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 $(D(v) \cap U(u)) - D(W) \neq \emptyset$; u is* **data flow dependent** *on v iff there is a sequence $v_1, v_2, \ldots, v_n$ 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 $i = 1, 2, \ldots, n-1$.*

The direct data flow dependence relation can be efficiently computed using a fast algorithm for the "reaching definitions" problem [Aho86]. The data flow dependence relation can then be efficiently computed using a fast algorithm for transitive closure [Aho74].

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 $(D(v_i) \cap U(v_{i+1})) - D(W_i) \neq \emptyset$ for $i = 1, 2, \ldots, n-1$. Such a walk is said to **demonstrate** the data flow dependence of u upon v.

Referring to the def/use graph of Figure 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 $v_1 v_2 v_3$; 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.

3.3 Syntactic Dependence

To evaluate uses of control and data dependence, it is necessary to consider *chains* of such dependences, that is, sequences of vertices such that each vertex in the sequence 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 *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 *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 [Berg85, Denn77, Ferr87, Horw88a, Kore87, Padu86, Weis84].

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

Definition 11 *Let $\mathbf{G} = (G, \Sigma, D, U)$ be a def/use graph, and let $u, v \in V(G)$. Vertex u is* **strongly syntactically dependent** *on vertex v iff there is a sequence $v_1, v_2, \ldots, v_n$ of vertices, $n \geq 2$, such that $u = v_1$, $v = v_n$, and for $i = 1, 2, \ldots, n-1$, either v_i is 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

efficiently computed, the weak and strong syntactic dependence relations can be efficiently computed using a fast algorithm for transitive closure.

Referring to the def/use graph of Figure 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.

4 Related Work

In this section we briefly survey the literature on uses of dependences in testing, debugging, and maintenance and on the semantic basis for these uses.

In software testing, a "code coverage criterion" is a rule, used for selecting test data, that is intended to ensure that certain portions of a program's code are "covered" or "exercised". Several coverage criteria have been defined that call for exercising the data flow dependences in a program [Fran87, Lask83, Ntaf84, Rapp85]; these are called "data flow coverage criteria" or simply "data flow criteria". The data flow criteria call for executing program walks[6] that demonstrate certain data flow dependences. One purpose for these criteria is the detection of statement faults. Data flow dependences involving potentially faulty statements are exercised in the hope that this will cause incorrect values to propagate via the sequence of assignments represented by a data flow dependence and, thus, cause observable failures.

For structured programs, Bergeretti and Carré [Berg85] define relations that are similar to strong syntactic dependence, and they suggest several uses for these relations, including testing and debugging.

Korel proposes two uses of program dependences [Kore87]. The first use is for detecting useless program statements; that is, statements that cannot influence the output of a program and can be removed without changing the function it computes. Korel claims that a statement is redundant if there is no output statement strongly syntactically dependent upon it. Korel's second use of program dependences is for determining which input variables in a program influence the value of a given output variable. Korel claims that an input variable influences an output variable if there is a strong syntactic dependence between the corresponding input and output statements.

In his work on program slicing, Weiser essentially claims that program dependences can 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. Weiser demonstrates how these dependences can be used to locate faults when debugging [Weis79, Weis82, Weis84]. Weiser 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 examining them the debugging process can be facilitated.

While most investigators who proposed uses for program dependences made no attempt to rigorously justify these uses, Weiser [Weis79] did recognize the need to rigorously justify the use of dependences in his program slicing technique, and attempted to provide such justification via both mathematical proofs and a psychological study. Unfortunately, the mathematical part of Weiser's work is flawed (see Section 7.3).

Recently, several papers have investigated the semantic basis for proposed uses of program dependences [Cart89, Horw88a, Horw88b, Reps89, Selk89]. Some of these papers address the use of dependences in software debugging and maintenance. Horwitz *et al* [Horw88a] present a theorem stating that two programs with the same dependences compute the same function. Reps and Yang [Reps89] use a version of this theorem to prove two theorems about program slicing, which are in turn used by Horwitz *et al* [Horw88b] to justify an algorithm for integrating versions of a program.

This paper differs in three respects from this 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 [Weis79], that makes them applicable to programs of any procedural programming language, in particular 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.

5 Semantic Dependence

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

We first informally define the auxiliary terms neces-

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

sary 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 $D(v)$. An interpretation of a def/use graph is similar to an interpretation of a program schema [Grei75, Mann74]. 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 assigment of values held by the variables of $U(v)$ (the variables used at v) just before the ith time v is visited during a computation. An execution history of a vertex in 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 12 *(Informal) A vertex u in a def/use graph $\mathbf{G}$ is* **semantically dependent** *on a vertex v of $\mathbf{G}$ if there are interpretations $\mathbf{I}_1$ and $\mathbf{I}_2$ of $\mathbf{G}$ that differ only in the function assigned to v, such that for some input, the execution history of u induced by $\mathbf{I}_1$ differs from that induced by $\mathbf{I}_2$.*[7]

For example, if the branch condition `X > Y` in the program of Figure 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 Figure 1 is semantically dependent on vertex v_2. As another example, if the statement `N := N - 1` in the program of Figure 2 were changed to `N := N - 2`, this would cause the **while**-loop not to terminate for the input $\mathtt{N} = 5$, and statement 6 would not be executed. Hence, this change demonstrates that vertex v_6 of the def/use graph of Figure 2 is semantically dependent on vertex v_5.

[7]The formal definition of semantic dependence, given in [Podg89], 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 say that (the semantics of) one program statement affects the execution behavior of another statement, this restriction is implied.

Note that a pair of execution histories of a vertex can differ in two ways: the histories can have corresponding entries that are unequal, or one history can be longer than the other. Informally, a semantic dependence is said to be **finitely demonstrated** if one of these two possibilities is demonstrated by finite portions of the computation sequences induced by a pair of interpretations and an input. Semantic dependence demonstrated by a pair of *halting* computations is, of course, finitely demonstrated.

6 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 "approximately" answered because all programs with a given def/use graph share certain semantic properties. To evaluate the use of dependence analysis in "approximately" answering the question above, we frame the question in terms of def/use graphs, by asking when one statement is semantically dependent on another. This leads to our main results (proofs are found in [Podg89]).

Theorem 2 *Let $\mathbf{G} = (G, \Sigma, D, U)$ 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.*

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

Theorem 4 *Let $\mathbf{G} = (G, \Sigma, D, U)$ 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.*

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

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

7 Implications of the Results

The results of Section 6 support the following 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 preclude all relationships between them that imply (non finitely-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 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 because strong syntactic dependence does not imply semantic dependence. Conclusion 3 follows from Theorem 4; any relationship between two statements 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" on (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 easily determined 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 conclusions 1–4 to evaluate existing uses of dependences in testing, debugging, and maintenance, and to suggest new ones. These conclusions address the semantic basis for certain uses of dependence analysis. The conclusions suggest that some proposed uses are mistaken, but provide partial justification, in terms of facilitating search, for other uses.

7.1 Dependence-Coverage Criteria

Since operator faults are one of the types of faults coverage criteria are intended to reveal, they are used here to illustrate coverage criteria that exercise dependences.

To help ensure that an operator fault is revealed, the data flow criteria exercise data flow dependences upon potentially faulty statements. This may cause erroneous values resulting from an operator fault to propagate via the sequence of assignments represented by a data flow dependence. In this way, a failure may be exposed. The data flow criteria differ with regard to the number of data flow dependences exercised, the types of data flow dependence exercised, and the number of walks executed that demonstrate a given data flow dependence. In the next few paragraphs, we discuss the significance of these differences for the detection of operator faults.

A data flow coverage criterion that exercises only a single direct data flow dependence upon each potentially faulty definition is the All-Defs criterion [Rapp85]. One might assume that exercising a single direct data flow dependence upon a definition with an operator fault is likely to cause an incorrect value to propagate to the dependent statement. However, by conclusion 4, a direct or indirect data flow dependence between two statements does not indicate that an operator fault at one of the statements can affect the execution history of the other. Thus, conclusion 4 controverts the use of data flow criteria like All-Defs, which exercise only a single data flow dependence per potentially faulty statement, for the detection of operator faults. It is possible that there is a *statistical* correlation between the existence of data flow dependences and the propagation of failures among the statements they relate; however, to the authors' knowledge, this has not been demonstrated.

The more demanding data flow criteria, such as Rapps and Weyuker's All-Uses and All-DU-Paths criteria [Rapp85], Laski and Korel's Strategy II [Lask83], and Ntafos's Required k-Tuples criteria [Ntaf84], call

for exercising *all* data flow dependences of certain types. This may indicate that the inventors of these criteria believe that these types of data flow dependence are necessary, but not sufficient, conditions for certain common types of failure propagation. This belief is compatible with conclusions 1–4. The All-Uses and All-DU-Paths criteria exercises all direct data flow dependences. The Required k-Tuples criteria exercise 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. Finally, Laski and Korel's Strategy II exercises all direct data flow dependences, but exercises them in combination.

Some of the data flow criteria considered here call for exercising multiple walks that demonstrate a given data flow dependence. Exercising any particular walk demonstrating a data flow dependence on a faulty statement does not ensure that a failure will be propagated, because the effect of the fault may be masked out. Presumably the inventors of the criteria requiring multiple walks wanted to decrease the likelihood that this would happen.

By conclusions 1–3, the fact that an operator fault at one statement can affect the behavior of another statement may involve control dependence, instead of just data flow dependence, as when an incorrect branch is taken by a conditional statement. Experience suggests that almost arbitrary chains of control and data flow dependences may be involved. This implies that almost any syntactic dependence not exercised by a coverage criterion might be the only one by which a given fault can be revealed. Hence, even the more demanding data flow coverage criteria do not ensure that all dependences are exercised that are relevant to the propagation of failures caused by operator faults. This is also true of those data flow criteria that include limited forms of control dependence coverage, such as the Required k-Tuples criteria. One might think that to remedy this weakness it is only necessary to exercise all syntactic dependences upon a potentially faulty definition. We suspect, however, that the number of dependences involved and the number of walks that must be executed to exercise them is too large for this strategy to be practical. Nevertheless, the syntactic dependences in a program provide a nontrivial bound on those statements that could be affected by an operator fault, and it should be possible to exploit this bound in developing testing methods. Successful use of this information presumably requires ways of filtering the syntactic dependences to be exercised and of selecting the test data to be used. Satisfying these requirements will undoubtedly entail more refined semantic analysis of dependences.

7.2 Anomaly Detection

A **program anomaly** is a textual pattern that is often evidence of a programming error [Fosd76] — for example, a variable being used before it has been defined. Korel's use of strong syntactic dependence to detect useless statements [Kore87] is a type of anomaly detection.

Korel did not prove his claim that if no output statement in a program is strongly syntactically dependent on a statement s then s cannot influence the program's output. Nevertheless, conclusion 3 supports a version of his informal claim: 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 non finitely-demonstrated semantic dependence, however. By conclusion 2, non finitely-demonstrated semantic dependence might not be accompanied by strong syntactic dependence. In the factorial program of Figure 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.

7.3 Debugging and Maintenance

In both software debugging and maintenance, it is necessary to determine when the semantics of one statement can affect the 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 thesis [Weis79], Weiser attempts to formally prove his claims about the relevance of program dependences to debugging, and he states a theorem[8] similar to Theorem 2. 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 in the first place, 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 by determining weak syntactic dependences, since the statement where the failure was observed is weakly syntactically dependent upon the faulty 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 7.2. Hence, Weiser's use of strong syntactic dependence in slicing is not justified in general, but weak syntactic dependence can be used instead.

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 modified statements are syntactically dependent on are relevant to the behavior of the modified statements.

8 Conclusion

In summary, we have shown that two generalizations of control and data dependence, called weak and strong syntactic dependence, are necessary conditions for certain interstatement relationships. Weak and strong syntactic dependences can be efficiently computed, and hence can be used to guide such activities as test data selection and code inspection. On the other hand, we have also shown that neither data flow nor syntactic dependence is a sufficient condition for such interstatement relationships, and this reflects negatively on the use of such dependences 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 testing and debugging are unjustified.

There are several possible lines of further investigation related to the use of program dependences in testing, debugging, and maintenance. The results of Section 6 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 software testing methods that exploit the fact that syntactic dependence can be used to bound the statements that can be affected by a fault. For example, Morell [More84], Richardson and Thompson [Rich88], and Demillo *et al* [DeMi88] propose test data selection methods based on determining conditions for erroneous program states to occur and then propagate to a program's output. These methods could exploit the dependence relations defined in this paper. A third possible line of investigation involves using more sophisticated semantic analysis techniques to complement dependence analysis.

[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.

References

[Aho74] Aho, A. V., Hopcroft, J. E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Aho86] Aho, A. V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Berg85] Bergeretti, J. F. and Carré, B. A. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, Jan. 1985.

[Cart89] Cartwright, R. and Felleisen, M. The semantics of program dependence. *SIGPLAN '89*, 1989.

[DeMi88] DeMillo, R.A., Guindi, D.S., King, K.N., McCracken, W.M., and Offutt, A.J. An extended overview of the Mothra software testing environment. *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, July 1988, pp. 142–151.

[Denn77] Denning, D. E. R. Certification of programs for secure information flow. *Communications of the ACM*, Vol. 20, No. 7, July 1977, pp. 504–513.

[Ferr87] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 5, July 1987, pp 319–349.

[Fosd76] Fosdick, L. D. and Osterweil, L. J. Data flow analysis in software reliability. *ACM Computing Surveys*, Vol. 8, No. 3, September 1976, pp. 306–330.

[Fran87] Frankl, P. G. The use of data flow information for the selection and evaluation of software test data. Doctoral Thesis, New York University, 1987.

[Grei75] Greibach, S. A. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer-Verlag, Berlin, 1975.

[Horw88a] Horwitz, S., Prins, J., and Reps, T. On the adequacy of program dependence graphs for representing programs. *Fifteenth ACM Symposium on Principles of Programming Languages*, ACM, New York, 1988.

[Horw88b] Horwitz, S., Prins, J., and Reps, T. Integrating non-interfering versions of programs. *Fifteenth ACM Symposium on Principles of Programming Languages*, ACM, New York, 1988.

[Kore87] Korel, B. The program dependence graph in static program testing. *Information Processing Letters 24*, Jan. 1987, pp. 103–108.

[Lask83] Laski, J. W. and Korel, B. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983, pp. 347–354.

[Mann74] Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[More84] Morell, L. J. A theory of error-based testing. Doctoral thesis, University of Maryland, 1984.

[Ntaf84] Ntafos, S. C. On Required Element Testing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, Nov. 1984, pp. 795–803.

[Padu86] Padua, D. A. and Wolfe, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1184–1201.

[Podg89] Podgurski, Andy. The significance of program dependences for software testing, debugging, and maintenance. Technical report, Computer and Information Science Department, University of Massachusetts, Amherst, 1989.

[Rapp85] Rapps, S. and Weyuker, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, Apr. 1985, pp. 367–375.

[Reps89] Reps, T. and Yang, W. The semantics of program slicing. Technical report, University of Wisconsin-Madison, 1989.

[Rich88] Richardson, D. J. and Thompson, M. C. The RELAY model of error detection and its application. *Proc. Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, 1988.

[Selk89] Selke, R. P. A rewriting semantics for program dependence graphs. *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989, pp. 12–24.

[Weis79] Weiser, M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Doctoral dissertation, University of Michigan, Ann Arbor, MI, 1979.

[Weis82] Weiser, M. Programmers use slices when debugging. *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 446–452.

[Weis84] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984, pp. 352–356.