Technical Report GIT-CERCS-03-10

Understanding Data Dependences in the Presence of Pointers*

Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold
College of Computing, Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA 30332
{orso,sinha,harrold}@cc.gatech.edu

Abstract

Understanding data dependences in programs is important for many software-engineering activities, such as program understanding, impact analysis, reverse engineering, and debugging. The presence of pointers, arrays, and structures can cause subtle and complex data dependences that can be difficult to understand. For example, in languages such as C, an assignment made through a pointer dereference can assign a value to one of several variables, none of which may appear syntactically in that statement. In the first part of this paper, we describe two techniques for classifying data dependences in the presence of pointer dereferences. The first technique classifies data dependences based on definition type, use type, and path type. The second technique classifies data dependences based on span. We present empirical results to illustrate the distribution of data-dependence types and spans for a set of real C programs. In the second part of the paper, we discuss two applications of the classification techniques. First, we investigate different ways in which the classification can be used to facilitate data-flow testing and verification. We outline an approach that uses types and spans of data dependences to determine the appropriate verification technique for different data dependences; we present empirical results to illustrate the approach. Second, we present a new slicing paradigm that computes slices based on types of data dependences. Based on the new paradigm, we define an incremental slicing technique that computes a slice in multiple steps. We present empirical results to illustrate the sizes of incremental slices and the potential usefulness of incremental slicing for debugging.

Keywords: Data dependences, pointers, program slicing, incremental slicing, data-flow testing, program comprehension.

1 Introduction

Understanding data dependences in programs is important for many software-engineering activities, such as program understanding, impact analysis, reverse engineering, and debugging. In fact, the effectiveness of such activities depends, to a large extent, on the availability of reliable information about dependences among program variables. Such dependences can be identified by computing definition-use (def-use) associations, which relate statements that assign values to variables to statements that use those values. The problem of computing def-use associations in the absence of pointers is relatively straightforward: in such cases, definitions and uses of variables can be identified by using only syntactic information, and def-use associations can be computed using a traditional data-flow analysis algorithm [3].

Unfortunately, traditional approaches for computing def-use associations are inadequate in the presence of pointers. The use of pointers can cause subtle and complex data dependences that can be difficult to understand. For example, an assignment made through a pointer dereference, in a language such as C,

*Earlier versions of the material presented in this paper appeared in the *Proceedings of the 9^{th} International Workshop on Program Comprehension* (May 2001) [36] and the *Proceedings of the International Conference on Software Maintenance* (November 2001) [37].

can assign a value to one of several variables, none of which may appear syntactically in that statement. Understanding the data dependences caused by such assignments is more difficult than understanding the dependences caused by direct (i.e., syntactic) assignments.

To assist software developers in the complex tasks of understanding data dependences, we have developed two techniques for classifying data dependences based on their characteristics. We have also investigated the application of the techniques for data-flow testing and debugging.

In the first part of this paper, we present our two techniques for classifying data dependences. The first technique classifies a data dependence based on the types of definition and use and the types of paths between the definition and the use. This technique distinguishes data dependences based on their strength and on the likelihood that a data dependence identified statically actually holds. It extends the classification presented by Ostrand and Weyuker [38] to provide a finer-grained and more general taxonomy of data dependences. The second technique classifies data dependences based on their spans. The span of a data dependence identifies the extent (or the reach) of that data dependence in the program, either at the procedure level or at the statement level. To compute and classify data dependences according to our classification schemes, we extend the traditional reaching-definitions algorithm.

The main benefit of these classification techniques is that they provide additional information about data dependences—information that can be used to compare, rank, prioritize, and understand data dependences, and can benefit software-engineering activities that use data dependences. In the first part of the paper, we also present empirical results to illustrate the distribution of types and spans of data dependences for a set of real C programs.

In the second part of the paper, we present two applications of the classification techniques.

First, we investigate how the classification techniques can be used to facilitate data-flow testing and verification. Although data-flow testing techniques [15, 41] have long been known, they are rarely used in practice, primarily because of their high costs [7]. The main factors that contribute to the unreasonably high costs are (1) the large number of def-use associations to be covered, a number of which may be infeasible,[1] (2) the difficulty of generating test inputs to cover the def-use associations, and (3) expensive program instrumentation required to determine the def-use associations that are covered by test inputs.

We investigate how classifying data dependences can help lower the costs of data-flow testing—by providing a way to order data dependences for coverage, to estimate the extent of data-flow coverage achieved through less-expensive testing, and to suggest the appropriate verification technique based on the types of data dependences that occur in the program. In the absence of information about data dependences, all data dependences are treated uniformly for data-flow testing. By providing information about various characteristics of a data dependence, the classification techniques can provide testers not only guidance in ordering data dependences for coverage, but also help in generating test inputs to cover them. We outline an approach that uses types and spans of data dependences to determine the appropriate verification technique for different data dependences and present empirical results to illustrate the approach.

Second, we present a new slicing paradigm that computes program slices [49] by considering only a subset of data dependences. This paradigm lets developers focus only on particular kinds of data dependences (e.g., strong data dependences) and provides a way to reduce the sizes of slices, thus making the slices more manageable and usable.

[1] A def-use association is *infeasible* if there exists no input to the program that causes that association to be covered.

Based on the new slicing paradigm, we present an incremental slicing technique that computes a slice in steps by incorporating different types of data dependences at each step. Consider, for instance, the use of slicing for program comprehension. When the developers are trying to understand just the overall structure of a program, they can ignore weaker data dependences and focus on stronger data dependences only. To do this, they can use the incremental slicing technique to start the analysis by considering only stronger data dependences, and then augment the slice incrementally by incorporating additional weaker data dependences. This approach lets the developers focus initially on a smaller, and thus potentially easier to understand, subset of the program and then consider increasingly larger parts of the program. Alternatively, for applications such as debugging, the developers may want to start focusing on weak, and therefore not obvious, data dependences. By doing this, they can identify subtle pointer-related dependences that may cause unforeseen behavior in the program.

To evaluate our incremental slicing approach, we implemented the technique, by extending the SDG-based approach for slicing [22, 42, 45], and performed two empirical studies. The first study shows the potential usefulness of the approach for reducing the fault-detection time during debugging. The second study shows that the results of incremental slicing generalize over more subjects, thus making the technique more generally applicable.

The main contributions of the paper are:

- Two techniques—one based on types and the other based on spans—for classifying data dependences in languages such as C.
- Empirical results that illustrate the occurrences of data-dependence types and spans for a set of real C programs.
- Application of the classification techniques to facilitate data-flow testing; empirical results to demonstrate how the classification can be used to estimate data-flow coverage and select the appropriate verification technique for data dependences.
- A new paradigm for slicing, in which slices are computed based on types of data dependences, and an incremental slicing technique that computes a slice in steps by incorporating additional types of data dependences at each step.
- Empirical studies that illustrate the sizes of incremental slices and the usefulness of incremental slicing for debugging.

The rest of the paper is organized as follows. In the next section, we present background material. In Section 3, we present our techniques for classifying data dependences in the presence of pointers; we also present empirical data to illustrate the distribution of data dependences for a set of C programs. In Section 4, we present two applications of the classification techniques. First, in Section 4.1, we discuss how the classification can be applied to data-flow testing. Second, in Section 4.2, we present a new slicing paradigm in which slices are computed based on data-dependence types; based on the paradigm, we present an incremental slicing technique. In Section 5, we discuss related work, and finally, in Section 6, we present conclusions and identify potential future work.

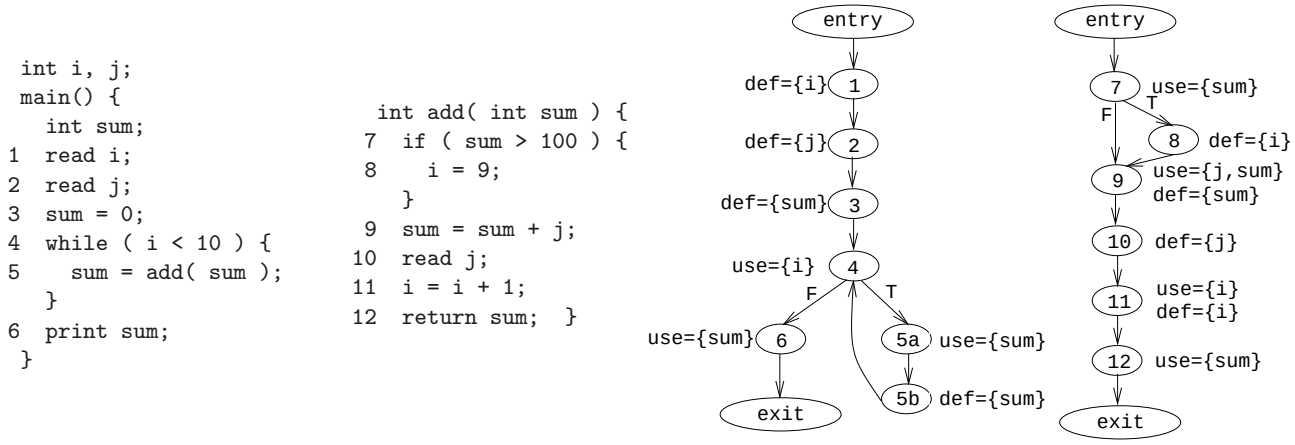

Figure 1: Program `Sum1` to illustrate definitions, uses, and data dependences (left); control-flow graphs for the program annotated with def and use sets (right).

2 Background

In this section, we present background material for the paper: data-flow analysis, alias analysis, data-flow testing, and program slicing.

2.1 Data-flow analysis

Data-flow analysis techniques require the control-flow relation of the program being analyzed. This relation can be represented in a control-flow graph. A *control-flow graph* (CFG) contains nodes that represent statements,[2] and edges that represent potential flow of control among the statements. In addition, the CFG contains a unique entry node and a unique exit node. For each call site, the CFG contains a call node and a return node. For example, Figure 1 presents program `Sum1` and the CFGs for the procedures in the program.

A statement *defines* a variable if the statement assigns a value to that variable. A statement *uses* a variable if the statement reads the value of that variable. For example, in `Sum1`, statement 1 defines variable `i` and statement 4 uses `i`; statement 9 uses `j` and `sum` and defines `sum`. To compute data dependences, the nodes in a CFG are annotated with two sets of variables: the *definition set*, $def(n)$, for a node n contains those variables that are defined at node n; the *use set*, $use(n)$ contains those variables that are used at node n. For example, in `Sum1`, def(9) = {`sum`} and use(9) = {`j`, `sum`}.

A *path* in a CFG is a sequence of nodes $(n_1, n_2, \ldots, n_k)$, $k \geq 0$, such that, if $k \geq 2$, for $i = 1, 2, \ldots, k-1$, (n_i, n_{i+1}) is an edge in the CFG. A *definition-clear path* (def-clear path) with respect to a variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that no node in $n_1, n_2, \ldots, n_k$ defines v. For example, in `Sum1`, (7, 9, 10, 11) is a def-clear path with respect to variable `i`, whereas, because of the definition of `i` at node 8, path (7, 8, 9, 10, 11) is not. A definition d_2 *kills* a definition d_1 if both d_1 and d_2 refer to the same variable v, and there exists a def-clear path with respect to v between d_1 and d_2. For example, the definition of `i` at node 11 kills the definition of `i` at node 8.

[2] A CFG can also be built at the basic-block level; in such a CFG, each node represents a sequence of single-entry, single-exit statements.

```
    int i;
    main() {
        int *p;
        int j, sum1, sum2;
 1.     sum1 = 0;
 2.     sum2 = 0;
 3.     read i, j;
 4.     while ( i < 10 ) {
 5.         if ( j < 0 ) {
 6.             p = &sum1;
            }
            else {
 7.             p = &sum2;
            }
 8.         *p = add( j, *p );
 9.         read j;
        }
10.     sum1 = add( j, sum1 );
11.     print sum1, sum2;
    }

    int add( int val, int sum ) {
        int *q, k;
12.     read k;
13.     if ( sum > 100 ) {
14.         i = 9;
        }
15.     sum = sum + i;
16.     if ( i < k ) {
17.         q = &val;
        }
        else {
18.         q = &k;
        }
19.     sum = sum + *q;
20.     i = i + 1;
21.     return sum;
    }
```

Figure 2: Program `Sum2`.

A *reaching-definition set*, $rd(j)$, defined with respect to a node j, is the set of variable–node pairs $< v, i >$ such that $v \in \text{def}(i)$ and there exists a def-clear path with respect to v from i to j. A *data dependence* is a triple (d, u, v), defined with respect to nodes d and u and variable v, such that $v \in \text{use}(u)$ and $< v, d > \in \text{rd}(u)$. A data dependence is also referred to as a *definition-use association* (def-use association or DUA). The computation of data dependences can be performed by first computing reaching definitions, and then examining, for each use, the reaching definitions for that use [3].

2.2 Alias analysis

In languages that contain usage of pointers, the computation of def-use associations requires the identification of alias relations. An *alias* occurs at a program point if two or more names refer to the same memory location at that point. An alias relation at program point n is a *may alias* relation if the relation holds on some, but not all, program paths leading up to n. An alias relation at point n is a *must alias* relation if the relation holds on all paths up to n. As an example, consider program `Sum2` (Figure 2).[3] In line 8, `*p` is a may alias for `sum1` and `sum2`, because it can refer to either `sum1` or `sum2`, depending on the path followed to reach statement 8 (i.e., depending on whether statement 6 or statement 7 is executed). *Alias analysis* or *points-to analysis* determines, at each statement that contains a pointer dereference, the set of memory locations that can be accessed through the dereference. For example, the alias set for `*p` at statement 8 contains two elements: `sum1` and `sum2`. A variety of alias-analysis algorithms have been presented in the literature; these algorithms vary in the efficiency and the precision with which they compute the alias relations (e.g., [4, 24, 27, 46]).

[3] `Sum2` is an extension of `Sum1` with the addition of pointers; it is overly complicated to illustrate our technique and the complex dependences that can be caused by pointers.

2.3 Data-flow testing

Data-flow testing techniques use data-flow relationships in a program to guide the selection of test inputs (e.g., [20, 26, 33, 41]). For example, the all-defs criterion [15, 41] requires the coverage of each definition in the program to some reachable use; the stronger all-uses criterion requires the coverage of each def-use association in the program. Other criteria require the coverage of chains (of different lengths) of data dependences [33].

2.4 Program slicing

A *program slice* of a program $\mathcal{P}$, computed with respect to a *slicing criterion* $< s, V >$, where s is a program point and V is a set of program variables, includes statements in $\mathcal{P}$ that may influence, or be influenced by, the values of the variables in V at s [49]. A program slice identifies statements that are related to the slicing criterion through transitive data and control dependences.[4]

Interprocedural slicing techniques based on the system-dependence graph (SDG) [22, 45] and data-flow equations [18, 49] form two alternative, general classes of slicing techniques. For this work, we extend the SDG-based slicing approach [22, 42, 45] (the approach based on data-flow equations [18, 49] could be extended similarly). The SDG-based approach precomputes all dependences and represents them in the SDG, and then computes slices using graph reachability.

A *system-dependence graph* (SDG) [22] is a collection of *program-dependence graphs* (PDG) [13]—one for each procedure—in which nodes represent statements or predicate expressions. Edges in the PDG represent data and control dependences. A *data-dependence edge* represents flow of data between statements; a *control-dependence edge* represents a control dependence of a statement on a predicate. Each PDG contains an *entry node* that represents entry to the procedure. To model parameter passing, an SDG associates formal parameter nodes with the entry node of each procedure: a *formal-in node* for each formal parameter of the procedure and a *formal-out node* for each formal parameter that may be modified [25] by the procedure. (Global variables are treated as parameters.) An SDG associates a set of actual parameter nodes with each call site in a procedure: an *actual-in node* for each actual parameter at the call site and an *actual-out node* for each actual parameter that may be modified by the called procedure.

To create parameter nodes, the SDG-construction algorithm [22] computes two sets of variables for each procedure: GMOD, which contains parameters and non-local variables that may be modified by the procedure, and GREF, which contains parameters and non-local variables that may be referenced by the procedure. For example, consider program `Sum1` (Figure 1). The GREF set for procedure `add` contains `i`, `j`, and `sum`; the GMOD set of `add` contains `i` and `j`. For each procedure P, the algorithm creates a formal-in node for each variable that appears in GMOD(P) or GREF(P), and a formal-out node for each variable that appears in GMOD(P). For a call site that calls procedure P, the algorithm creates an actual-in node for each variable that appears in GMOD(P) or GREF(P), and an actual-out node for each variable that appears in GMOD(P).[5]

[4] A statement s is *control dependent* on a predicate p if, in the CFG, there are two edges out of the node for p such that by following one edge, the node for s is definitely reached, whereas by following the other edge, the node for s may not be reached. [13]

[5] The algorithm also creates a formal-out node (and a corresponding actual-out node at each call site) for the return value from a function.

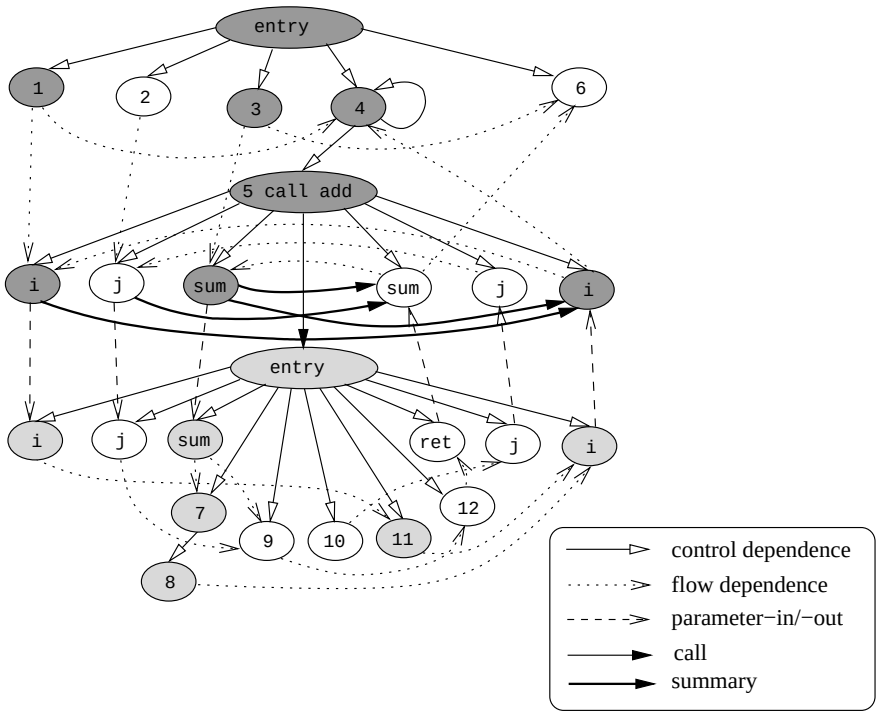

Figure 3: System-dependence graph for `Sum1`. The shaded nodes are included in the slice for $< \texttt{5b}, \{\texttt{i}\} >$; the darker nodes are included during the first phase of the slicing algorithm, whereas the lighter nodes are included during the second phase.

An SDG connects PDGs at call sites. A *call edge* connects a call node to the entry node of the called procedure's PDG. Parameter-in and parameter-out edges represent parameter passing: *parameter-in edges* connect actual-in nodes to formal-in nodes, and *parameter-out edges* connect formal-out nodes to actual-out nodes.

Horwitz, Reps, and Binkley [22] compute interprocedural slices by solving a graph-reachability problem on an SDG. To restrict the computation of interprocedural slicing to paths that correspond to valid call–return sequences, an SDG uses summary edges to represent the transitive flow of dependence across call sites caused by data dependences, control dependences, or both. A *summary edge* connects an actual-in node and an actual-out node if the value associated with the actual-in node may affect the value associated with the actual-out node.

Figure 3 shows the SDG for `Sum1`. Each parameter node in the figure is labeled with the variable to which the node corresponds; the formal-out node for the return value is labeled 'ret'. The call nodes in the SDG have actual-in nodes associated with them for actual parameters or global variables that may be referenced or modified by the called procedure; the call nodes also have actual-out nodes for parameters and global variables that may be modified by the called procedure. Similarly, the entry nodes have formal-in and formal-out nodes associated with them. For example, procedure `add` references global variables `i` and `j` and formal parameter `sum`; it modifies `i` and `j`. Therefore, the entry node corresponding to `add` has formal-in and formal-out nodes for these variables associated with it; the entry node also has a formal-out node for the return value. The call node that corresponds to the call to `B` has three actual-in and three actual-out nodes associated with them. At each call site, the SDG contains summary edges for transitive data and/or

control dependences. For example, because the value of `sum` after the call in line 5 depends on the values of `sum` and `j` prior to the call, the SDG contains summary edges from the actual-in nodes for `sum` and `j` to the actual-out node for `sum` at that call site.

In their original work on SDG-based slicing, Horwitz, Reps, and Binkley [22] described an algorithm based on attribute grammars for computing summary edges. Later, Reps and colleagues [42] presented a faster, simpler algorithm based on graph reachability. That algorithm extends paths backwards along data-dependence and control-dependence edges, starting at all formal-out nodes. The algorithm iteratively determines whether formal-in nodes are reachable from formal-out nodes and computes summary edges.

The interprocedural backward slicing algorithm consists of two phases. The first phase traverses backwards from the node in the SDG that represents the slicing criterion along all edges except parameter-out edges, and marks those nodes that are reached. The second phase traverses backwards from all nodes marked during the first phase along all edges except call and parameter-in edges, and marks reached nodes. The slice is the union of the marked nodes.

For example, consider the computation of a slice for $< 5, \{\texttt{i}\} >$; this slicing criterion is represented in the SDG of Figure 3 by the actual-out node for `i` at the call site to `add`. In its first phase, the algorithm adds to the slice the nodes shown in darker shading in Figure 3. In its second phase, the algorithm adds the nodes shown in lighter shading in the figure.

Unlike static slicing techniques, which consider dependences that can occur in any execution of a program, dynamic slicing techniques [2, 23] consider only those dependences that occur in a particular execution of the program; dynamic slicing techniques ignore those static dependences that do not occur in that execution. A dynamic slicing criterion contains, apart from the program point and the set of variables, an input to the program.

3 Data Dependences in the Presence of Pointers

In the presence of pointer dereferences, it may not be possible to identify unambiguously the variable that is actually defined (or used) at a statement containing a definition (or use). To account for such effects, we developed a technique for classifying data dependences into different types; this technique extends the classification presented by Ostrand and Weyuker [38]. In Section 3.2, we present a second technique for classifying data dependences, based on their spans. In Section 3.3, we briefly describe our algorithms for computing types and spans. In Section 3.4, we present empirical results to illustrate the occurrences of different data-dependence types and spans in practice.

3.1 Classification of data dependences based on types

We classify data dependences based on the types of definitions and uses, and the types of paths from definitions to uses.

3.1.1 Types of definitions and uses

In the presence of pointers, memory locations can be accessed not only directly through variable names, but also indirectly through pointer dereferences. Unlike a direct access, an access through a pointer dereference

can potentially access one of several memory locations. For example, in program `Sum2`, statement 2 defines `sum2` through direct access. However, statement 8 defines variables through indirect access—the variable that is actually defined at statement 8 is the variable to which `p` points at that statement. Depending on the execution path to that statement, `p` can point to different variables: if the predicate in statement 5 is true, `p` points to `sum1` at statement 8, whereas if the predicate in statement 5 is false, `p` points to `sum2` at that statement. Thus, statement 8 can potentially define either `sum1` or `sum2`.

The traditional notion of definitions and uses does not differentiate direct accesses from indirect accesses, and can thus provide misleading information about the occurrences of those accesses. In the example just described, statement 2 defines `sum2` on all executions, whereas statement 8 can define `sum1` on some executions and `sum2` on other executions. Thus, the execution of statement 8 is not sufficient for either of these definitions to occur, which has important implications. For example, consider a code based-testing technique that targets memory accesses for coverage. To cover a direct access, the technique can target the statement containing the access; however, to cover an indirect access, the technique must target not only the statement containing the access, but also statements that establish alias relations for the indirect access. Thus, distinguishing direct accesses from indirect accesses provides useful information for understanding how execution of statements can result in memory accesses.

To distinguish different ways in which memory can be accessed in the presence of pointers, we define three types of memory accesses: direct, single alias, and multiple alias. A *direct access* involves no pointer dereference. A *single-alias access* occurs through a dereference of a pointer that can point to a single memory location. A *multiple-alias access* occurs through a dereference of a pointer that can point to multiple memory locations. A direct access results in a *definite definition* or *definite use* of the memory location being accessed, whereas a single-alias or a multiple-alias access results in a *possible definition* or *possible use* of the memory location being accessed.[6]

Thus, based on the types of definitions and uses, it is possible to have nine types of data dependences. Figure 4 shows the CFGs for the procedures in `Sum2` and lists, for each node in the CFG, the definite and possible definitions and uses that occur at that node.

3.1.2 Types of paths from definitions to uses

Types of definitions and uses provide information about the occurrences of the definition and the use for a data dependence. However, this is insufficient information about the occurrence of the data dependence—the classification provides no information about the paths over which the definition may propagate to the use. Such paths can contain definite or possible redefinitions (or kills) of the relevant variable, which can prevent the definition from propagating to the use. Failure to distinguish possible kills along a path can provide misleading information about paths between definitions and uses. A path that contains only possible kills can be identified as containing no kills.[7] Thus, a definition may actually not reach a use along such a path, although the analysis would rule out such an occurrence.

We classify paths from definitions to uses based on the occurrences of definite, possible, or no kills along the paths. Let (d, u, v) be a data dependence. In the absence of pointer dereferences, it is sufficient to classify

[6] A single-alias access to a memory location is considered a possible definition or use of the accessed memory location because of the limitation of static analysis in approximating the dereferenced memory locations.

[7] A conservative data-flow analysis requires such an assumption [3].

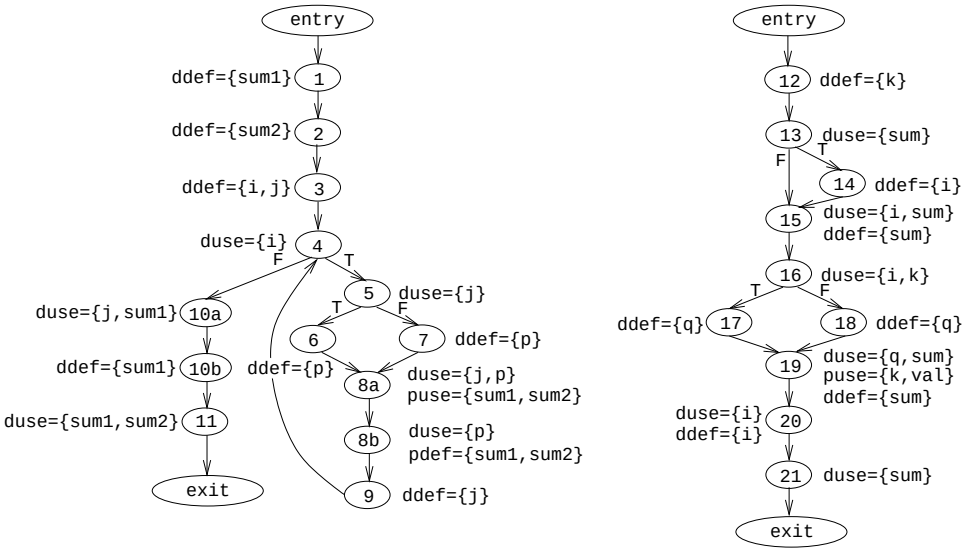

Figure 4: Control-flow graphs for the procedures in `Sum2` (Figure 2) with definite and possible definition and use sets at each node.

---

each path π from d to u into one of two types, based on whether the definition at d is killed along path π. However, the presence of single-alias and multiple-alias accesses introduces an additional category in which π can be classified: a definition may be possibly killed along π. Thus, in the presence of pointers, we classify π into one of three types.

A *definite def-clear path* with respect to variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that no node in $n_1, n_2, \ldots, n_k$ contains either a definite or a possible definition of v. For example, in program `Sum2`, path (1, 2, 3, 4, 10a) is a definite def-clear path with respect to variable `sum1`.

A *possible def-clear path* with respect to variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that there exists at least one n_i, $1 \leq i \leq k$, that contains a possible definition of v, but no node in $n_1, n_2, \ldots, n_k$ contains a definite definition of v. For example, in program `Sum2`, the path (8a, 8b, 9, 4, 10a) is a possible def-clear path with respect to variable `sum1`, because node 8b contains a possible definition of `sum1` and no node in the path contains a definite definition of `sum1`.

A *definite killing path* with respect to variable v is a path $(i, n_1, n_2, \ldots, n_k, j)$ such that there exists at least one n_i, $1 \leq i \leq k$, that contains a definite definition of v. For example, in program `Sum2`, the path (10a, 10b, 11) is a definite killing path with respect to variable `sum1`, because node 10b contains a definite definition of `sum1`.

We associate colors green, yellow, and red with the three types of paths: green with definite def-clear paths, yellow with possible def-clear paths, and red with definite killing paths. The analogy with a traffic light is helpful in getting an intuitive idea of the meaning of such paths: a green path for memory location v propagates definitions of v from the beginning of the path to the end of the path; a yellow path for v may or may not propagate definitions of v; and a red path for v does not propagate definitions of v to the end of the path.

Typically, for a data dependence, there is a set of paths from the definition to the use. Because each path

Table 1: Seven rd types based on the occurrences of green (definite def-clear), yellow (possible def-clear), and red (definite killing) paths in the set of paths from definitions to uses.

Occurrence of green, yellow, and red paths	Rd type
{green}	G
{green, red}	GR
{green, yellow}	GY
{green, yellow, red}	GYR
{yellow}	Y
{yellow, red}	YR
{red}	R

Table 2: Data dependences, with their types, that occur in program Sum2.

Data dependence	Type	Data dependence	Type	Data dependence	Type
(1, 8a, sum1)	(D, MA, GY)	(7, 8a, p)	(D, D, GR)	(14, 16, i)	(D, D, G)
(1, 10a, sum1)	(D, D, GY)	(8b, 8a, sum1)	(MA, MA, GY)	(14, 20, i)	(D, D, G)
(2, 8a, sum2)	(D, MA, GY)	(8b, 10a, sum1)	(MA, D, GY)	(15, 19, sum)	(D, D, G)
(2, 11, sum2)	(D, D, GY)	(8b, 8a, sum2)	(MA, MA, GY)	(17, 19, q)	(D, D, G)
(3, 4, i)	(D, D, G)	(8b, 11, sum2)	(MA, D, GY)	(18, 19, q)	(D, D, G)
(3, 15, i)	(D, D, GR)	(9, 5, j)	(D, D, GR)	(19, 21, sum)	(D, D, G)
(3, 16, i)	(D, D, GR)	(9, 8a, j)	(D, D, GR)	(20, 4, i)	(D, D, GR)
(3, 20, i)	(D, D, GR)	(9, 10a, j)	(D, D, GR)	(20, 15, i)	(D, D, GR)
(3, 5, j)	(D, D, GR)	(10a, 11, sum1)	(D, D, G)	(20, 16, i)	(D, D, GR)
(3, 8a, j)	(D, D, GR)	(12, 16, k)	(D, D, G)	(20, 20, i)	(D, D, GR)
(3, 10a, j)	(D, D, GR)	(12, 19, k)	(D, MA, G)		
(6, 8a, p)	(D, D, GR)	(14, 15, i)	(D, D, G)		

in this set can be classified as green, yellow, or red, the set of paths can be classified in seven ways, depending on the occurrence of green, yellow, and red paths in the set. We refer to the classification of the set of paths from definition to use as the *reaching-definition type* or the *rd type*. Table 1 lists seven possible rd types for a data dependence. The seventh type consists only of red paths; in this case, because the definition is killed along all paths from the definition to the use, the definition and the use do not form a data dependence. Thus, there are six rd types of interest.

For example, in program Sum2, for data dependence (1, 8a, sum1), the rd type is GY, whereas, for data dependence (3, 8a, j), the rd type is GR; for data dependence (3, 4, i), the rd type is G.

3.1.3 Types of data dependences

Based on the types of definitions and uses and on the rd types, a data dependence can be classified into one of 54 types (nine combinations of definition and use types, together with six rd types). Table 2 lists the data dependences, along with their types, that occur in program Sum2; the type of a data dependence is listed using the triple (def type, use type, rd type). To succinctly identify definition and use types, we use the abbreviations D for direct, SA for single-alias, and MA for multiple-alias accesses. For example, data dependence (1, 8a, sum1) has type (D, MA, GY), which corresponds to a direct definition, multiple-alias use, and {green, yellow} paths between the definition and the use.

3.2 Classification of data dependences based on spans

Although types of data dependences are useful for understanding how a data dependence occurs, they do not provide information about parts of a program that may need to be examined to understand a data dependence. To provide such information, we present an alternative way to classify data dependences based

on spans. Intuitively, the span of a data dependence is the extent, or the reach, of the data dependence: it is the portion of the program over which the data dependence extends and, therefore, includes parts of the program that may need to be examined to understand the data dependence. Like data-dependence types, data-dependence spans can be used to group and order data dependences. Spans can potentially be useful for understanding data dependences and for generating test inputs to cover data dependences. A span can be defined at different levels of granularity, such as procedures and statements.

A *procedure span* of a data dependence (d, u, v) is a set of triples $<$ *proc*, *occ*, *color* $>$; the set contains an element for the procedure that contains the definition d, an element for the procedure that contains the use u, and an element for each procedure that contains a definite or possible kill for the data dependence. Each element in a procedure span is a triple:

- *proc* identifies the procedure.
- *occ* specifies the occurrence type for the procedure: whether the procedure contains definition, use, or kill for the data dependence, or any combination of the three. The possible values for *occ* are `d` for definition, `u` for use, `k` for kill, or any combination of the three: `du`, `dk`, `uk`, or `duk`.
- *color* identifies the types of kills that occur in the procedure, if any. The possible values for *color* are: (1) `G`, if the procedure contains no kills (that is, the occurrence type is `d`, `u`, or `du`), (2) `R`, if the procedure contains only definite kills, (3) `Y`, if the procedure contains only possible kills, and (4) `YR`, if the procedure contains definite and possible kills.

The size of a procedure span for an intraprocedural data dependence is one; for an interprocedural data dependence, the size of a procedure span can vary from one to the number of procedures in the program. For example, the procedure span for data dependence (9, 5, `j`) in program `Sum2` is {<`main`, `duk`, `R`>}; for data dependence (1, 10a, `sum1`), the procedure span is {<`main`, `du`, `G`>, <`add`, `k`, `Y`>}.

A statement span is defined similarly; its elements correspond to statements instead of procedures. A *statement span* of a data dependence (d, u, v) is a set of quadruples $<$ *proc*, *stmt*, *occ*, *color* $>$; the set contains an element for the statement that contains the definition d, an element for the statement that contains the use u, and an element for each statement that contains a definite or possible kill for the data dependence. Each element in a statement span is a quadruple:

- *proc* and *stmt* identify the procedure and the statement, respectively.
- *occ* specifies the occurrence type for the statement: whether the statement contains definition, use, or kill for the data dependence, or any combination of the three.
- *color* identifies the types of kills that occur in the statement. The possible values for *color* are: (1) `G`, if the statement contains no kills (that is, the occurrence type is `d`, `u`, or `du`), (2) `R`, if the statement contains a definite kill, and (3) `Y`, if the statement contains a possible kill for the data dependence.

The size of a statement span can vary from one to the number of statements in the program. For example, the statement span for data dependence (9, 5, `j`) in program `Sum2` is {<`main`, 9, `dk`, `R`>, <`main`, 5, `u`, `G`>}; for data dependence (1, 10a, `sum1`), the statement span is {<`main`, 1, `d`, `G`>, <`main`, 10a, `u`, `G`>, <`add`, 19, `k`, `Y`>}.

The definition of span can be extended to incorporate other types of information. For example, for each occurrence of a procedure (or statement) that contains a possible definition, the span can be expanded to include the procedures (or statements) that introduce the alias relations relevant for that possible definition.

Data-dependence spans are related to data-dependence types. For example, the rd type for a data dependence determines the occurrences of colors in the span for that data dependence. For rd type `G`, at most two elements can appear in a span. Spans provide a measure of the complexity of a data dependence that is different than the measure provided by types; spans and types can be used in conjunction to obtain a better and more complete estimate of the complexity of data flow in a program. For example, data dependences can first be classified based on types; then, for each type, the data dependences can be classified based on procedure or statement spans. In Section 4, we illustrate how types and spans can be leveraged for different applications of data dependences.

3.3 Computation of data-dependence types and spans

To compute rd types, we extend the traditional algorithm for computing reaching-definitions to propagate two additional sets of data-flow facts at each statement. The first set contains the possible definitions that reach a statement; the second set contains the killed definitions that reach a statement. Like the traditional algorithm, the extended algorithm computes the three sets iteratively until the sets converge.

To compute and classify data dependences, we extend Harrold and Soffa's algorithm [21], which computes interprocedural data dependences in two phases. In the first phase, the extended algorithm analyzes each procedure and computes information that is local to the procedure. The local information consists of intraprocedural data dependences (along with their types) and the information that is required for the interprocedural phase. In the second phase, the algorithm (1) builds a representation, called the interprocedural flow graph, and (2) traverses the graph to compute and classify interprocedural data dependences. Reference [35] contains a detailed description of the algorithm.

To compute spans of interprocedural data dependences, we use the extended interprocedural-flow-graph-based algorithm [35]. First, during the construction of the interprocedural flow graph, the algorithm computes summary information about each procedure; the summary information for a procedure P contains, for each data dependence that reaches from the entry of P to the exit of P, the definite and possible kills that occur in P or in some procedure directly or indirectly called in P. Second, during the traversal of the interprocedural graph to compute a data dependence, the algorithm propagates information about definite and possible kills.

3.4 Empirical results

Our example (`Sum2`, Figure 2) shows that the presence of pointers and pointer dereferences can cause a number of different types of data dependences to occur: seven different types of data dependences occur in `Sum2`. To investigate how these data-dependence types occur in practice in real programs, we performed an empirical study. We implemented the reaching-definitions algorithm using the Aristotle analysis system [19]. For alias information, we used the alias analysis described in Reference [29]; that implementation is based on the PROLANGS Analysis Framework (PAF) [40].

3.4.1 Goals and method

The overall goal of our empirical study was to examine the occurrences of different data-dependence types and spans in real C programs. We used 13 C programs, drawn from diverse sources, as subjects for the empirical study. Table 3 describes the subject programs and lists the number of non-comment lines of code in each program.

Table 3: Programs used for the empirical studies reported in the paper.

Subject	Description	LOC
armenu	ARISTOTLE analysis system user interface	6067
bison	Parser generator	5542
dejavu	Interprocedural regression test selector [43]	3166
flex	Lexical analyzer generator	8264
larn	A dungeon-type game program	7715
lharc	Compress/extract utility	2500
mpegplay	MPEG player	12354
mpegplayer	Another MPEG player	5380
sed	GNU batch stream editor	5418
space	Parser for antenna-array description language	6199
T-W-MC	Layout generator for cells in circuit design	21379
unzip	Zipfile extract utility	2834
xearth	Display program for a shaded image of the earth in the X root window	21333

Table 4: The number of data dependences and data-dependence types computed for the subjects.

	Intraprocedural		Interprocedural		Total	
Subject	DUAs	Types	DUAs	Types	DUAs	Types
armenu	2948	10	3139	22	6087	22
bison	9527	10	17423	8	26950	11
dejavu	2475	6	788	11	3263	11
flex	7344	13	6411	17	13755	18
larn	10638	20	182819	20	193457	22
lharc	2336	15	1281	19	3617	23
mpegplay	45429	17	462277	26	507706	30
mpegplayer	14821	24	77706	28	92527	35
sed	35193	12	23424	21	58617	23
space	18100	14	10898	15	28998	17
T-W-MC	48051	21	92011	20	140062	23
unzip	2128	15	1497	22	3625	23
xearth	3311	13	2200	11	5511	16

For each subject program, we computed intraprocedural and interprocedural data dependences and their types. First, we examined the number of different types of data dependences that occurred in each subject and the frequency of those occurrences. Second, we studied the distribution of interprocedural data dependences based on their procedure spans.

3.4.2 Results and analysis

Occurrences of data-dependence types. We begin by examining the number of data dependences and the number of data-dependence types computed for the subject programs. Table 4 shows the number of intraprocedural and interprocedural data dependences for each subject. The table also shows the number of data-dependence types that occurred among the intraprocedural and interprocedural data dependences and in total for each subject. The data in the table show that several types of data dependences can occur: the number of data-dependence types that appears in a subject varies from 11 to as many as 35. Programs that have a large number of data dependences, such as `larn`, `mpegplay`, `mpegplayer`, and `T-W-MC`, also have many different types of data dependences. Even program such as `lharc` and `unzip`, that have relatively fewer data dependences, have several types of data dependences occurring in them. For most of the subjects, more types occurred among the interprocedural data dependences than among intraprocedural data dependences.

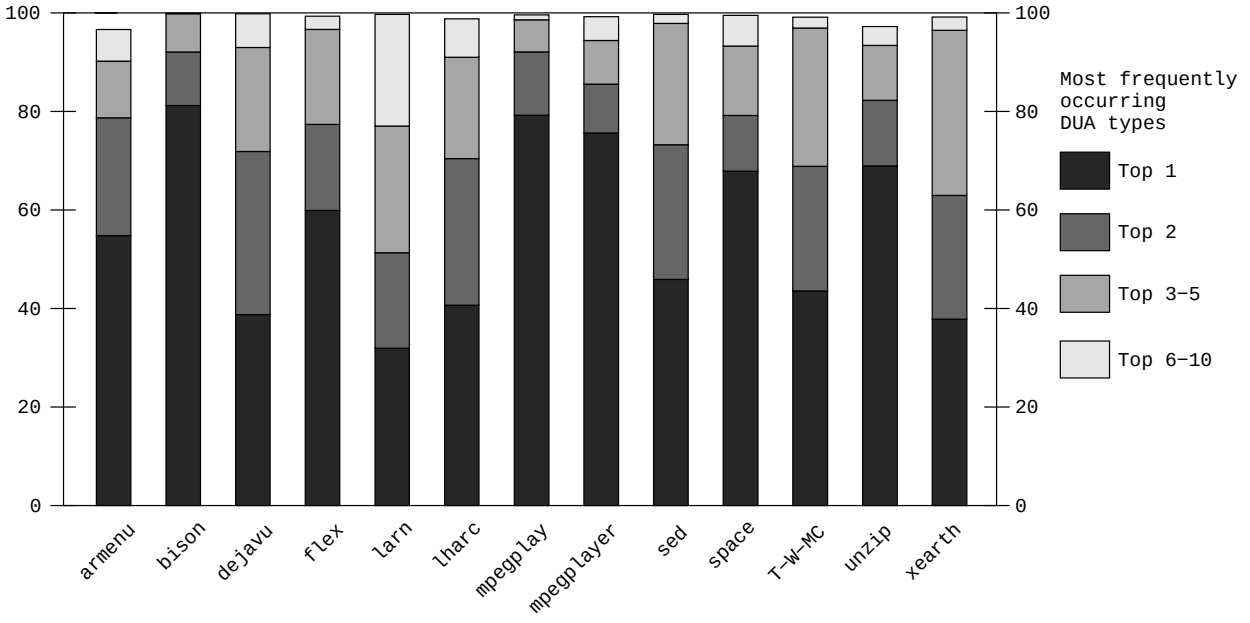

Figure 5: Percentage of data dependences accounted for by the most-frequently-occurring data-dependence types: top 1, top 2, top 3–5, and top 6–10.

Table 5: The top-three most-frequently-occurring types of data dependences.

Subject	Top 1	Top 2	Top 3
armenu	(D, D, GR)	(D, D, G)	(D, D, GYR)
bison	(D, D, GR)	(D, D, G)	(SA, SA, GY)
dejavu	(D, D, GR)	(D, D, G)	(SA, SA, GY)
flex	(D, D, GR)	(D, D, G)	(SA, SA, GY)
larn	(D, D, Y)	(D, SA, Y)	(SA, SA, Y)
lharc	(D, D, GR)	(D, D, G)	(D, D, Y)
mpegplay	(SA, SA, Y)	(SA, SA, GY)	(D, D, GR)
mpegplayer	(SA, SA, Y)	(D, D, GR)	(MA, MA, Y)
sed	(SA, SA, Y)	(D, D, GR)	(SA, SA, GY)
space	(D, D, Y)	(D, D, G)	(SA, SA, Y)
T-W-MC	(SA, SA, Y)	(SA, SA, GY)	(D, D, GR)
unzip	(D, D, GR)	(D, D, G)	(SA, SA, GY)
xearth	(D, D, GR)	(D, D, G)	(SA, SA, Y)

Given that a number of different types of data dependences can occur, next, we examine the frequency with which the types occur. Figure 5 presents data about the percentage of data dependences that were accounted for by the 10 most-frequently-occurring data-dependence types. The figure contains one segmented bar per subject; the vertical axis represents the percentage of data dependences in the subjects. The segments within each bar partition the 10 most-frequently-occurring data dependences into four sets—top 1, top 2, top 3–5, and top 6–10—which represent, respectively, the most-frequently-occurring, the second most-frequently-occurring, the third to fifth most-frequently-occurring, and the sixth to tenth most-frequently-occurring data-dependence types. For example, for `armenu`, the most-frequently-occurring data-dependence type accounted for nearly 55% of the data dependences; the next most-frequently-occurring data-dependence type accounted for another 24% of the data dependences.

Table 6: The number of occurrences of each data-dependence type.

RD type	(D, D)	(D, SA)	(D, MA)	(SA, D)	(SA, SA)	(SA, MA)	(MA, D)	(MA, SA)	(MA, MA)
G	37469	127	24	528	3074	1	0	3	26
GY	21163	10845	80	13973	129576	737	49	739	2882
GR	124141	135	0	430	4	0	0	0	0
GYR	1341	440	4	456	184	0	1	0	18
Y	84178	38195	36	15357	583066	2150	4	728	11200
YR	354	335	1	29	80	0	6	0	6

The data in the figure show that, consistently across the subjects, a few types account for a majority of data dependences. For all subjects except larn, the top five data-dependence types account for more than 90% of the data dependences. The number of data dependences of the most-frequently-occurring type vary from 32% for larn to 81% for bison. Thus, although a large number of different data-dependence types occur (Table 4), few of those types occur in large numbers and the remaining types occur in very small numbers. For example, although 30 types of data dependences occur in mpegplay, only 10 of them account for over 99% of data dependences; the remaining 20 types together account for less than 1% of the data dependences. Similarly, for T-W-MC, 10 of the 23 types that occur in that subject account for 99% of the data dependences.

Table 5 lists the three most-frequently-occurring data-dependence types in the subjects. (D, D, GR) is the type that occurs most commonly in the table: it is the most-frequently-occurring type in six of the subjects, the second most-frequently-occurring type in two subjects, and the third most-frequently-occurring type in another two subjects. (D, D, GR) does not appear in the top three types for only three of the subjects. (D, D, G) is the second most-frequently-occurring type in eight of the subjects. (D, D, G) and (D, D, GR) are the simplest of the data-dependence types because they involve no pointer dereferences at the definition or the use, or in the paths between the definition and the use. Thus, the predominant occurrences of such types in a program indicates that the program manipulates simple data structures and has relatively simple data-flow complexity. This is true of programs such as armenu, bison, dejavu, flex, lharc, unzip, and xearth, as also confirmed by our manual inspection of these subjects. Other subjects, such as mpegplay, mpegplayer, and T-W-MC, manipulate complex data structures and, thus, have more complex data-dependences types appearing predominantly in them; (SA, SA, Y) is the most-frequently-occurring type in those three subjects.

Types in which the definitions or uses involve multiple-alias accesses do not appear prominently in Table 5. In fact, such types occur only once in the table: in mpegplayer, the third most-frequently-occurring type is (MA, MA, Y).

Another pattern evident in the data in Table 5 is that, for each type listed in the table, except one ((D, SA, Y) for larn), the access type at the definition is the same as the access type at the use. This may indicate a pattern in the way data dependences occur in C programs.

Next, we examine, for each data-dependence type, the number of times it occurs over all subjects; Table 6 presents this data. The data in the table show that those types in which one access, either at the definition or at the use, is multiple-alias and the other access is direct (Columns 3 and 7) occur in very small numbers. Other data dependences that involve a multiple-alias access (Columns 6, 8, and 9) also occur less frequently. Data dependences involving a multiple-alias access (Columns 3, 6, 7, 8, and 9) occur predominantly with rd

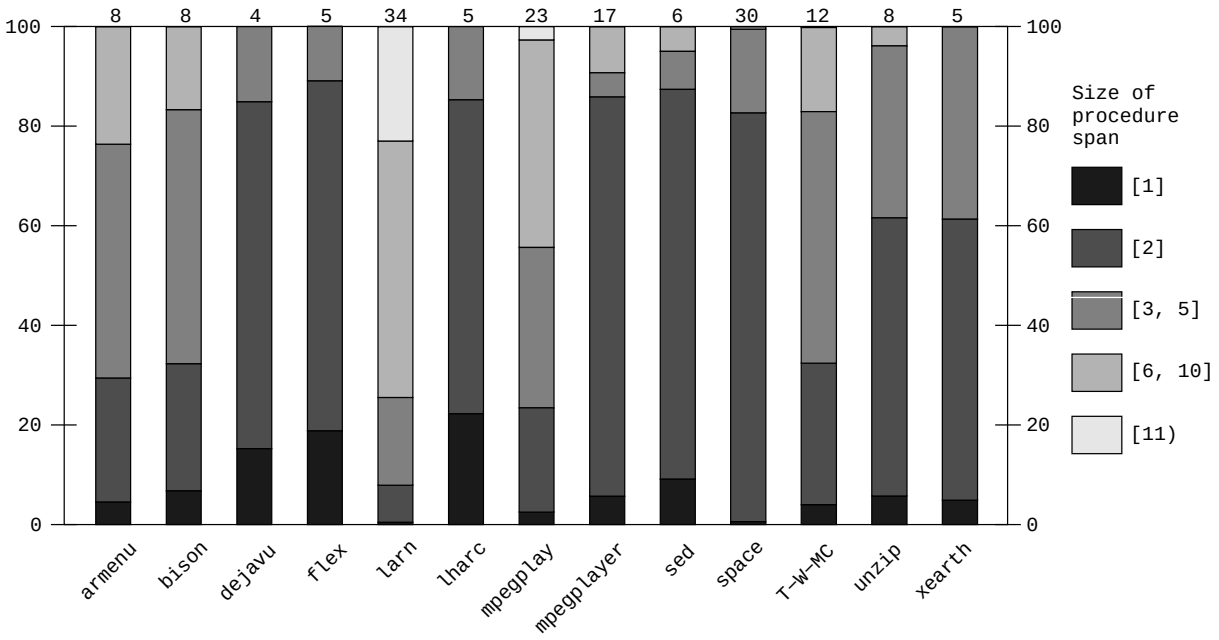

Figure 6: Distribution of interprocedural data dependences based on the number of elements in the procedure spans. Each segment represents the percentage of interprocedural data dependences that spanned a particular range of procedures. The number at the top of each bar is the size of the largest procedure span for that subject.

types `GY` and `Y` (Rows 2 and 5); they occur in negligible numbers with other rd types. This may indicate another pattern in the usage of pointers in C programs. Another significant pattern in the data is that rd types that involve a definite kill (i.e., rd types that include a red path), shown in Rows 3, 4, and 6, occur mostly in data dependences that involve a direct definition or a direct use.

Spans of interprocedural data dependences. Finally, we examine, for each interprocedural data dependence, the number of procedures that appeared in the procedure span of that dependence. As mentioned earlier, the size of the procedure span of each intraprocedural data dependence is one; thus, it needs not be examined.

Figure 6 presents the distribution of interprocedural data dependences based on the sizes of the procedure spans. Each segmented bar in the figure represents 100% of the interprocedural data dependences in that subject; the segments represent the percentages of interprocedural data dependences in the subject that spanned different numbers of procedures. The number at the top of each bar is the size of the largest procedure span for that subject. The figure illustrates that the number of data dependences that span a single procedure is not negligible; such data dependences occur because of successive calls to the same procedure, such that a definition from one call reaches a use in the next call.

For all subjects except `larn` and `mpegplay`, most of the data dependences have a procedure span of five or less. For six of the subjects, more than 80% of interprocedural data dependences have a span of one or two. For another two subjects, `unzip` and `xearth`, more than 60% of data dependences have a span of one or two. Spans larger than five occur in significant numbers in `armenu`, `bison`, `larn`, `mpegplay`, and `T-W-MC`. Spans larger than 10 appear in five subjects; but they appear in large numbers in only one, `larn`, in which 23% of

the interprocedural data dependences have spans that include more then 10 procedures. The largest span also occurs in `larn`—it includes 34 of the 179 procedures in that subject. The largest span, in terms of the percentage of procedures included in the span, occurs in `space`—it includes 30 (22%) of the 136 procedures in the program.

3.4.3 Discussion

The results of this study indicate several patterns. One pattern is that, although a number of different types of data dependences can occur in real C programs, not all types occur in equally significant numbers. A consistent result is that most of the data dependences fall predominantly into a few types; these few types can account for up to 90% or more of the data dependences in the programs. Examining the most-frequently occurring data-dependence types can help the developers to infer the overall data-flow complexity of programs. This information can be leveraged, for instance, when testing a program: programs with relatively simple data dependences are likely to be suitable for data-flow testing; whereas program with complicated data dependences may be more suitable for alternative verification techniques, such as software inspection. (We further discuss this application of our classification in Section 4.1.)

Another pattern observable in the data is that multiple-alias accesses occur predominantly with rd types `Y` or `GY`; data dependences with such accesses rarely have all green paths or a red path between the definitions and the uses (Table 6). The study of such patterns in a program can help the developers to get an overall view of the data-flow structure of a program. For example, consider a program in which each data dependence that consists of a definite definition and a definite use includes no yellow path between the definition and the use; such a program likely manipulates primarily statically-allocated data structures.

Overall, the results of the study show that a number of data-dependence types occur in the subjects, which is an adequate reason to investigate how activities that use data dependences can benefit from such information. In the next section, we present two applications that leverage such occurrences of different data-dependence types.

4 Applications of Data-Dependence Classification

The classification of data dependences can be used for several different applications. For example, data-dependence types can be used to define new data-flow testing criteria that target specific types of data dependences for coverage [38]; data dependences can also be ordered or prioritized for coverage based on their types. For another example, data-dependence types can be used to support impact analysis by focusing the analysis on specific types of data dependences. Data-dependence types can also be used for identifying parts of the code where subtle and possibly unforeseen data dependences require careful software inspections. In short, any activity that uses data-dependence information may benefit from such a classification. The primary benefit is that the classification lets such activities compare, group, rank, and prioritize data dependences, and to process various data dependences differently, based on their types, instead of processing all data dependences in the same way.

To support this claim, in this section, we present two applications of data-dependence classification. First, we investigate how the classification can be used to facilitate data-flow testing. Then, we present how the classification can be applied to program slicing, for use in activities such as debugging.

4.1 Data-flow testing

Data-flow testing techniques have long been known [15, 41]; these techniques provide better coverage of the elements of a program than other code-based testing techniques such as statement testing (i.e., coverage of each statement in a program) and branch testing (i.e., cover each conditional branch in a program) [12, 34, 41]. Previous research has also shown that data-flow testing can be more effective at detecting faults than branch testing [14, 16]. However, despite their apparent strengths and benefits, data-flow testing techniques are rarely used in practice, primarily because of their high costs [7]. As mentioned in Section 1, the main factors that contribute to this unreasonably high costs are (1) the large number of test requirements (or data dependences) to be covered, a number of which may be infeasible, (2) the difficulty of generating test inputs to cover the test requirements, and (3) expensive program instrumentation required to determine the data dependences that are covered by test inputs.

In the absence of information about data dependences, all data dependences must necessarily be treated uniformly for data-flow testing. The tester has no knowledge of the different costs associated with covering different data dependences; thus, the tester has no guidance in trying to order or prioritize data dependences for coverage to meet the constraints of time and cost. Moreover, in the absence of such information, the number of data dependences is the only measure for determining the viability of using data-flow testing for a program; the tester has no guidance in deciding whether alternative verification techniques, such as code inspection, may be more appropriate than testing. In the next three subsections, we discuss how the classification techniques can help the tester in ordering data dependences for coverage and generating test data to cover them (Section 4.1.1), estimating data-flow coverage from existing test suites (Section 4.1.2), and determining the appropriate verification technique for data flow (Section 4.1.3). In Section 4.1.4, we outline an approach that uses types and spans to determine the verification strategy for data dependences and present empirical results to illustrate the approach.

4.1.1 Ordering data dependences for coverage and generating test data

Ostrand and Weyuker [38] define new data-flow testing criteria that are designed to cover different types of data dependences. They discuss how their classification of data dependences can be used to order data dependences, on the basis of strength of the relationships, for coverage. Similarly, our classification provides a systematic way of grouping data dependences and prioritizing them for coverage.

Data dependences can be ordered based on types of definitions and uses, types of rd paths, or a combination of the two. The ordering can be based on the expected ease of covering the data dependences. For example, we expect data dependences with direct definitions and uses to be easier to cover than those with multiple-alias definitions and uses. To cover direct definitions and uses, it is sufficient to cover the statements in which the accesses occurs. In contrast, for multiple-alias definitions and uses, not only must the statements containing the definitions and uses be reached, they must also access the same memory location. Thus, to cover such definitions and uses, the statements that establish the alias relations must also be covered. Similarly, different rd types have different levels of complexity associated with them for coverage. Green and red paths provide definite information—they either propagate or do not propagate definitions each time that they are executed. In contrast, yellow paths provide information that is uncertain—they can propagate definitions on certain executions and not on others. Therefore, intuitively, we expect covering a

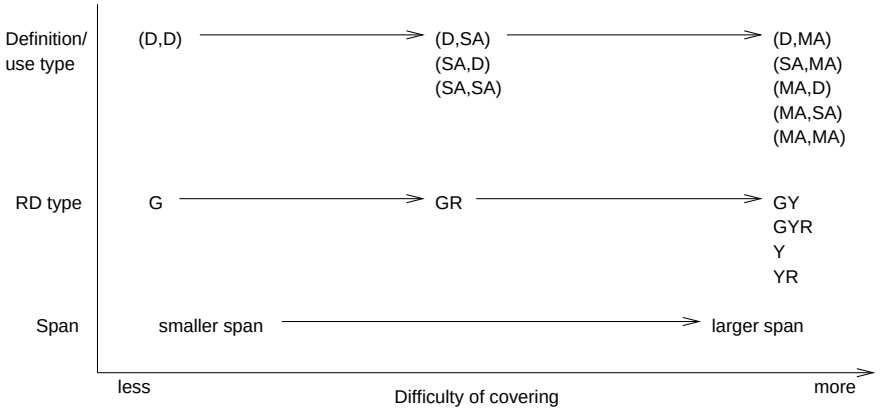

Figure 7: Relative expected difficulties of covering different types of definitions, uses, rd types, and spans.

data dependence with rd type `G` to be much easier than covering a data dependence with rd type `Y` or `YR`. In the latter case, not only must a yellow path be executed, but also the correct alias relations must hold along the path so that the definition propagates to the use.

Information about spans can be combined with information about types to further divide data dependences into subgroups. Data dependences with larger spans will generally be more difficult to cover than those with smaller spans. For example, data dependences with spans greater than five will typically be difficult to cover because the interactions involve several procedures; occurrences of possible definitions in these procedures will further complicate covering the data dependences. Thus, such data dependences can be scheduled for coverage later in the testing process, if sufficient time and resources permit them to be covered; or, such data dependences may not be targeted for coverage at all. Figure 7 summarizes the relative expected difficulties of covering different types of definitions, uses, rd types, and spans.

Once data dependences have been ordered for coverage, the classification can also aid with generating test data to cover the dependences. Using types, along with statement and procedure spans, can guide the tester in identifying statements that must be reached and those that should be avoided. Moreover, data-dependence spans can be extended to provide additional support for test-data generation. For example, the information can be extended to include alias information and alias-introduction dependences. At each statement that appears in a statement span and has color yellow associated with it, the span can be extended to include (1) the number of aliases at that statement, and (2) the statements that introduce the alias relations for that statement. The alias information could be computed using an approach similar to the one used by Pande, Landi, and Ryder to compute conditional reaching definitions [39]. This extended span information would enable the tester to navigate from such statements to the alias-introduction sites and better understand the conditions that must be satisfied to cover a data dependence.

4.1.2 Estimating data-flow coverage achieved through less-expensive testing

The classification of data dependences can be used to determine the percentage of data dependences that may be covered through less-expensive testing, such as statement or branch testing. The extent of data-flow coverage attained through less-expensive testing can be a useful measure of the adequacy of testing and of the additional cost of performing data-flow testing. The coverage of a large percentage of data

dependences increases the testers' confidence in the adequacy of testing using weaker criteria. On the one hand, it indicates to the testers that significant additional coverage of data dependences may not be attained through data-flow testing. On the other hand, it also indicates that data-flow coverage may be attained at a lower cost—by generating test data, and selectively instrumenting, for only the (few) remaining data dependences. In general, the classification can be used to guide the testers in measuring what proportion of the task of data-flow testing has already been completed and what remains to be done.

The classification can be used to estimate data-flow coverage in two ways.

First, the classification can be used to estimate statically, given coverage of all statements or branches, the data dependences that are also definitely covered. This applies to those data dependences that have direct definitions and uses and whose rd types are `G`. For such data dependences, covering the definition statement and the use statement suffices to cover the data dependences. A subset of this group of data dependences—those in which either the definition dominates the use or the use postdominates[8] the definition—can be covered simply by targeting either the definition statement or the use statement for coverage.

Thus, a test suite developed for statement coverage also covers all data dependences of type (`D`, `D`, `G`) in which either the definition dominates the use or the use postdominates the definition. The remaining data dependences of type (`D`, `D`, `G`) can be covered by developing test inputs to traverse the definition and use statements; we call this criteria *def-use coverage*. Def-use coverage is less expensive than data-dependence coverage in both the effort required to generate test inputs and the amount of instrumentation required to determine coverage.

Second, the classification can be used to infer, from coverage data gathered using instrumentation for def-use coverage, the data dependences that are covered in addition to those that were targeted for coverage by the test suite. To do this, the tester computes the statement spans of the remaining data dependences and orders them by the size of the span, to first consider data dependences with smaller spans. Next, the tester checks whether the coverage data for any test input includes the definition and use statements for a data dependence, but excludes the kill statements for the data dependence. If this is the case, the data dependence is covered by the corresponding test input. Note that this check can also be performed using procedure spans. To avoid iterating through all the remaining data dependences, the tester can set a threshold value for the span size and consider only data dependences with spans smaller than the threshold.

4.1.3 Determining the appropriate verification technique for data flow

The classification can also be used to determine the appropriate verification technique for the data flow occurring in a program. Not all data dependences are equivalent in terms of their complexity or the expected effort required to generate test data for them. Some data dependences, such as those that contain yellow paths between definitions and uses and span multiple procedures, may be too complicated to verify through testing. For such data dependences, testing may not be an effective verification technique; alternative verification techniques, such as code inspection, may be more appropriate. Other data dependences may be more suitable for verification through testing. In the absence of information about types and spans of data dependences, testers have no guidance in determining the appropriate verification technique for different data dependences.

[8] A statement s_i *dominates* a statement s_j if each path from the beginning of the program to s_j goes through s_i. A statement s_i *postdominates* a statement s_j if each path from s_i to the end of the program goes through s_j.

Table 7: The criteria used to determine the appropriate verification technique—testing or inspection—for different types of data dependences.

RD type	(D, D)	(D, SA)	(D, MA)	(SA, D)	(SA, SA)	(SA, MA)	(MA, D)	(MA, SA)	(MA, MA)
G	statement/def-use coverage	testing							
GY	testing if procedure span ≤ 3, otherwise inspection								
GR	testing if procedure span ≤ 4, otherwise inspection								
GYR	testing if procedure span ≤ 3, otherwise inspection								
Y	testing if procedure span ≤ 3, otherwise inspection								
YR	testing if procedure span ≤ 3, otherwise inspection								

4.1.4 Empirical results

To illustrate how the classification of data dependences can be applied, in practice, to data-flow testing, we conducted a case study using our subjects.

Goals and method. The overall goal of the study was to investigate whether the classification can be used to support data-flow testing. The steps that we used in the case study are as follows. First, for each subject, we determined the percentage of data dependences that would be covered by statement coverage. Then, we determined the additional data dependences that would be covered by def-use coverage. Next, we ordered the remaining data dependences by types and, within types, by spans. We then partitioned this set into those that could be targeted for coverage and those whose complexity would make test-data generation very difficult. To partition the data, we selected threshold values based on the complexity rankings shown in Figure 7.

As mentioned previously, data dependences of type (D, D, G) are implied by statement and def-use coverage. The coverage of remaining data dependences with rd types G requires the coverage of definition and use statements, ensuring that the memory locations being accessed at the definition and use statements are the same. By definition, such data dependences have a maximum procedure span of two. We expect the generation of test inputs for covering such data dependences to be easier than the generation of inputs for dependences that have a red or a yellow path between the definition and the use. Next, we considered data dependences with rd types GR; for such data dependences, each path from the definition to the use is definite def-clear or definite killing. Thus, intuitively, generating test data for such data dependences should be easier than generating data for those in which a yellow path appears between the definition and the use. For such data dependences, we set a threshold of four for the procedure span: dependences with procedure spans of four or less could be considered for coverage but those with spans greater than four would likely be too complicated.

We used a threshold of three for the remaining data dependences, whose rd types included a yellow path. Because generating test data in the presence of a yellow path can, in general, be more challenging, we used a smaller threshold value for such data dependences. Table 7 lists the criteria that we used in the case study to determine the appropriate verification technique.

Note that the values we selected are just one reasonable, possible set of values; the rationale for the selection of the thresholds is that they can be varied for different programs, based on the resources available for testing and on the testers' knowledge about the complexity of generating test data for dependences spanning multiple procedures. Thus, for some programs, the thresholds that we chose may be too high,

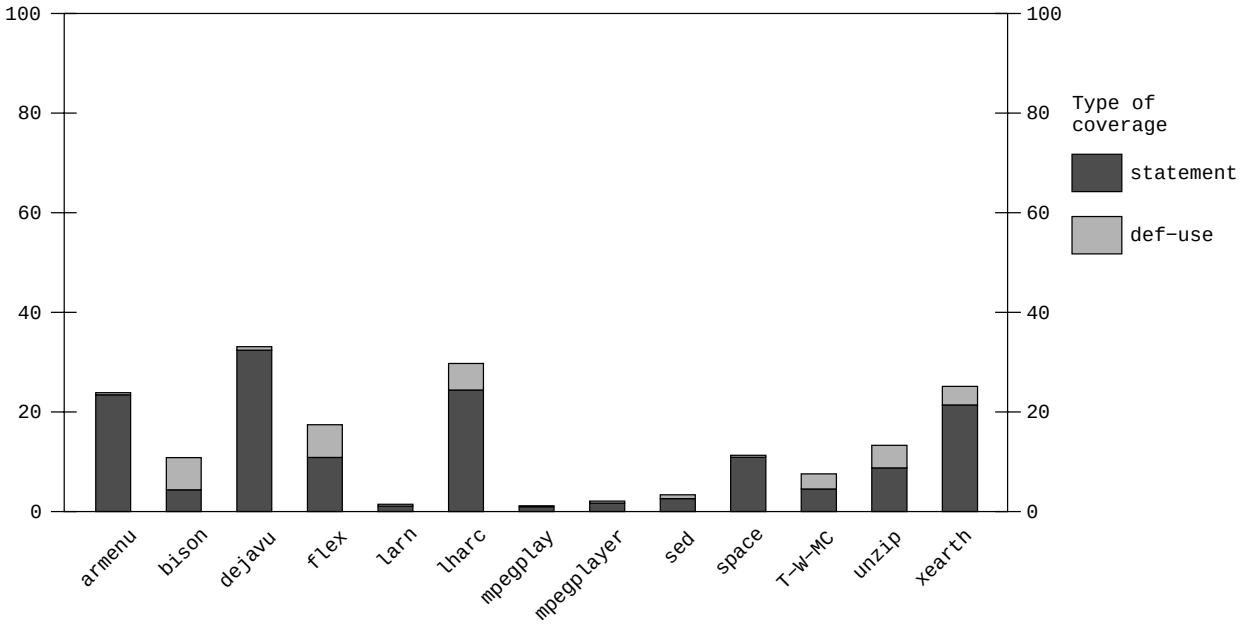

Figure 8: Percentage of data dependences that are covered by statement coverage and def-use coverage.

whereas, for others, they may be low.

Finally, for the data dependences that were outside our thresholds—for whom we deemed data-flow testing as being not practical and, thus, requiring an alternative verification technique—we computed the combined span of the data dependences. By computing the combined span, we were able to examine whether such complicated data dependences cluster in certain parts of the program or spread all over the program. The combined span identifies the parts of the program that would need to be examined during inspection.

Results and analysis. Figure 8 presents, for each subject, the percentage of data dependences that are covered by statement coverage and def-use coverage. Each segmented bar represents the percentage of data dependences of type (D, D, G). The darker segment within a bar represents those data dependences that are covered by a test suite that provides 100% statement coverage. These data dependences are a subset of the data dependences of type (D, D, G)—the subset in which either the definition dominates the use or the use postdominates the definition.[9] The percentage of data dependences covered by statement coverage varies from less than 1% for `larn` and `mpegplayer` to more than 25% for `dejavu` and `lharc`.

Def-use coverage covers a noticeably larger number of data dependences than statement coverage for six subjects; for the remaining subjects, def-use coverage increased the coverage of data dependences marginally. Thus, this data indicates that most of the data-dependences of type (D, D, G) occur intraprocedurally, and that for most of them, covering either the definition or the use suffices to cover the data dependence.

Figure 9 presents the data for determining the appropriate verification technique for the remaining data

[9]Because of the limitations of our analysis tools, we could compute intraprocedural dominance only. Therefore, the values represented by the darker segments represent intraprocedural dependences only; they are lower that what they would be had we analyzed interprocedural data dependences also. However, even with interprocedural analysis, the height of each bar would remain unchanged because each bar represents the total percentage of data dependences of type (D, D, G). The additional analysis can only cause the darker segment to occupy a larger proportion of each bar.

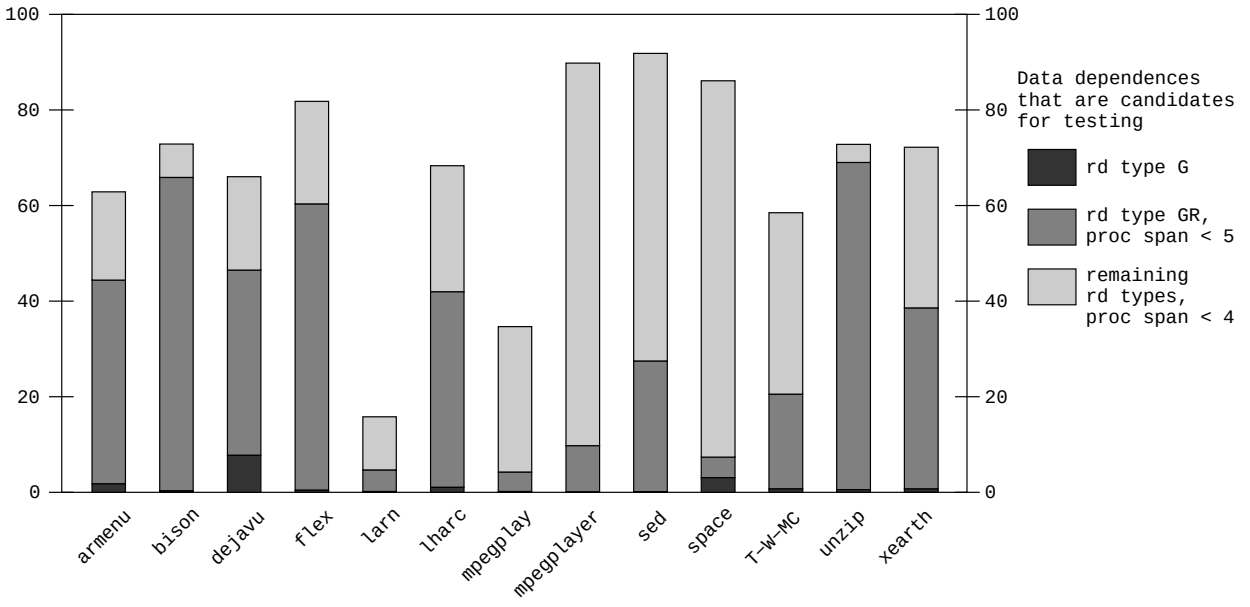

Figure 9: Percentage of remaining data dependences of different types and procedure spans that can be candidates for data-flow testing.

dependences—those that are covered by neither statement coverage nor def-use coverage. We order the remaining data dependences by types and, within types, by procedure spans. First, we consider remaining data dependences with rd type `G`. For such data dependences, the definition or the use (or both) involves a non-direct access. Therefore, to cover such data dependences, the definition and use statements must be reached and both statements must access the same memory location. With the exception of `dejavu`, such data dependences occur in very few numbers; covering them raises the total data-flow coverage of the programs marginally. For `dejavu`, the total of number of data dependences covered at this step is more than 40%; for other subjects, this percentage varies from under 2% to over 25%. In the second step, we consider data dependences with rd types `GR` and whose procedure spans are less than five. This step includes a large percentage of the data dependences in `armenu`, `bison`, `dejavu`, `flex`, `lharc`, `unzip`, and `xearth`. However, for other subjects, such as `larn`, `mpegplay`, and `space`, this step includes less than 5% of data dependences. This step increases the data-flow coverage to over 60% for seven subjects and over 25% for another two subjects. However, for two of the remaining four subjects, `mpegplayer` and `space`, data-flow coverage remains below 20%; and for the other two, `larn` and `mpegplay`, it remains less than 7%.

In the final step, we consider the remaining data dependences; all of these data dependences have at least one yellow path between the definition and the use. For such data dependences, we considered procedure spans of three or less. A majority of the data dependences in `mpegplayer`, `sed`, and `space` are included in this step. For nine of the subjects, this step increases the number of data dependences considered for testing to over 95% and, for another two subjects, to over 80%. However, for `larn` and `mpegplay`, the data-flow coverage is below 40%.

Table 8 lists the remaining data dependences in each subject; such data dependences can be candidates for verification using code inspection. Their complexity, both in terms of their types and their spans, makes

Table 8: Remaining data dependences that are candidates for verification through inspection and the cumulative procedure spans of those data dependences.

Subject	Number of data dependences		Cumulative procedure span	
armenu	807	13.3%	37	39.0%
bison	4385	16.3%	48	36.7%
dejavu	26	0.8%	5	5.5%
flex	100	0.7%	21	15.0%
larn	159983	82.7%	159	54.5%
lharc	69	1.9%	16	18.0%
mpegplay	325717	64.2%	72	51.4%
mpegplayer	7467	8.1%	28	26.4%
sed	2804	4.8%	24	31.8%
space	745	2.6%	95	69.3%
T-W-MC	47442	33.9%	137	60.9%
unzip	138	3.8%	17	41.5%
xearth	145	2.6%	10	9.9%

them extremely difficult to be verified through testing. Generating test data for such data dependences, if at all possible, may not be worth the time and effort that it would require. Table 8 also lists the cumulative procedure spans of the remaining data dependences. For subjects such as `larn`, `mpegplay`, and `T-W-MC`, that have a large number of remaining data dependences, the data dependences together span more than half the procedures in those subjects. These are the procedures that would have to be examined during inspection to verify those data dependences. For `space`, which has relatively few remaining data dependences, the dependences span a considerable percentage of the program—more than 69% of the procedures.

Again at this step, a subset of the data dependences can be selected, based on types or spans or both, for verification through inspection. This would, in turn, reduce the parts of the program that would need to be examined during inspection.

Discussion. In our empirical study, we have outlined an approach that can be used to select data dependences and order them, for coverage, based on an estimate of the ease of covering them. The approach that we have outlined and presented in the study is one possible instance of the general approach; in practice, it can be modified to suit the particular program being tested and the extent of data-flow coverage and verification desired for the program. The starting point for the approach is to determine the data-flow coverage attained from existing test suites. Next, if def-use coverage can provide significant additional coverage, test inputs can be developed, and the program instrumented, for def-use coverage. Finally, for the remaining data dependences, the appropriate verification technique can be selected. If, at any point in the process, the desired level of data-flow verification is attained, the process can be terminated. The results of the study show that the approach can be practical and effective, by actually providing useful information to the tester.

Note that, in our study, we selected testing and software inspection as the appropriate verification techniques for simple and complex data dependences, respectively. However, other techniques may prove to be more effective in verifying these kinds of data dependences. One important characteristic of our approach is that it is not tied to a specific technique or set of techniques: it provides a general way to group data dependences and to select different verification techniques for different groups.

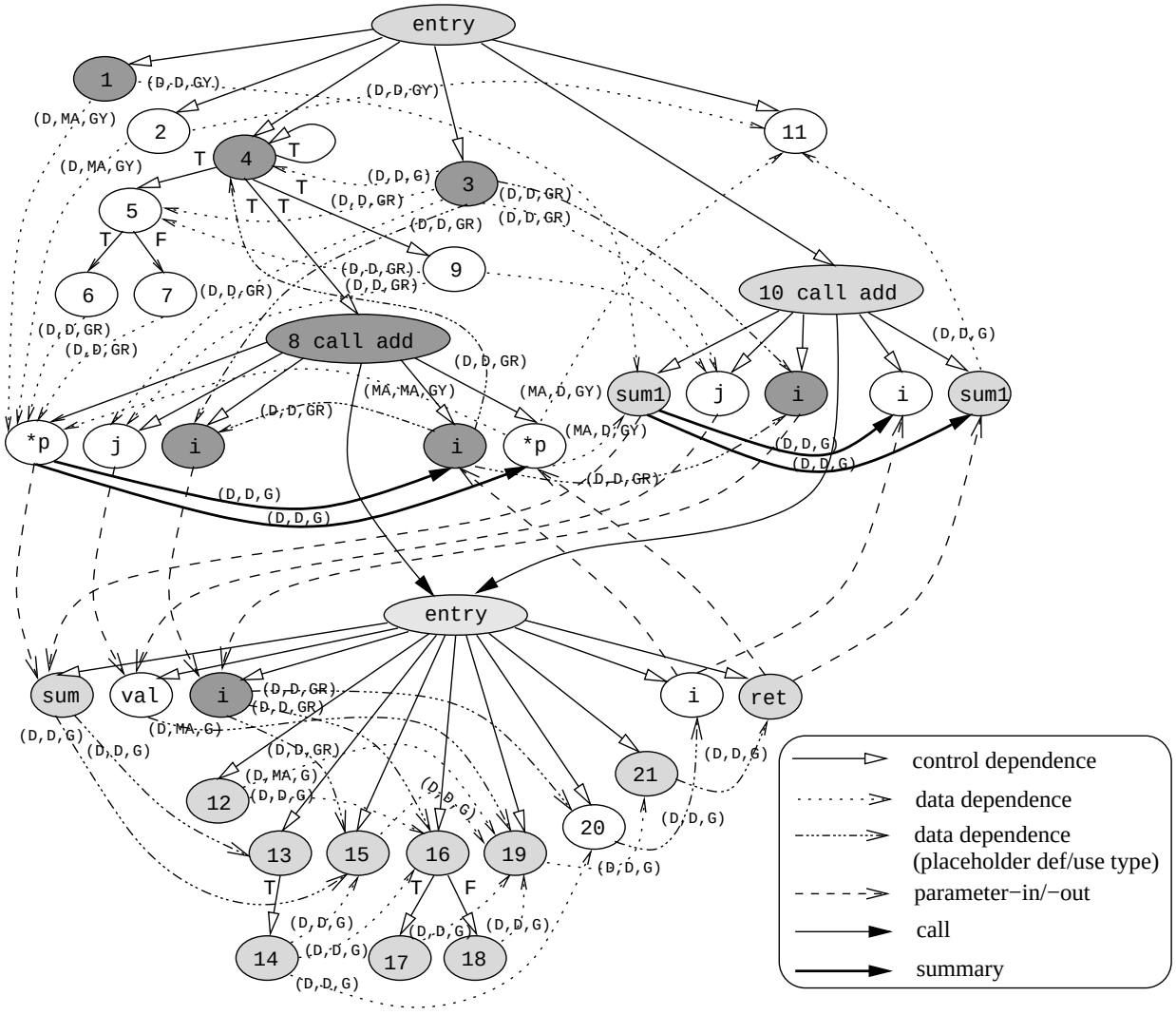

Figure 10: System-dependence graph for `Sum2` to support slicing using types of data dependences. The nodes shown in the lighter shade are included in the slice for $< 10, \{\texttt{sum1}\}, \{(\texttt{D}, \texttt{D}, \texttt{G})\} >$. The nodes shown in the darker shade are additional nodes included in the slice for $< 10, \{\texttt{sum1}\}, \{(\texttt{D}, \texttt{D}, \texttt{G}), (\texttt{D}, \texttt{D}, \texttt{GY}), (\texttt{D}, \texttt{D}, \texttt{GR})\} >$.

4.2 Incremental slicing based on data-dependence types

Traditional slicing techniques (e.g., [18, 22, 49]) include in the slice all statements that can affect the slicing criterion through direct or transitive control and data dependences. Such techniques compute a slice by computing the transitive closure of all control dependences and all data dependences starting at the slicing criterion. The classification of data dependences into different types leads to a new paradigm for slicing, in which the transitive closure is performed over only the specified types of data dependences, rather than over all data dependences. In the next two subsections, we describe this paradigm (Section 4.2.1) and present the algorithm for computing slices in the paradigm (Section 4.2.2). Based on the new paradigm, we describe an incremental slicing technique (Section 4.2.3). Finally, we present empirical results to illustrate the usefulness of incremental slicing for debugging (Section 4.2.4).

4.2.1 New slicing paradigm

In the new slicing paradigm, a *slicing criterion* is a triple $< s, V, T >$, where s is a program point, V is a set of program variables referenced at s, and T is a set of data-dependence types. A *program slice* contains those statements that may affect, or be affected by, the values of the variables in V at s through transitive control or specified types of data dependences. Slices can be computed in this new paradigm using either the SDG-based approach [22, 42, 45] or the data-flow-based approach [18, 49].

To compute slices in the new paradigm, using the SDG-based approach, we extend both the SDG and the SDG-based slicing algorithm.

Extensions to the SDG. We extend the SDG in two ways. First, we annotate each data-dependence edge with the type of the corresponding data dependence. The traditional SDG does not distinguish data dependences based on their types and, therefore, does not contain such annotations. To illustrate, Figure 10 presents the SDG for `Sum2`. Each data-dependence edge in the figure is labeled with the type of that data dependence. For example, the data-dependence edge from node 1 to the actual-in node for `*p` at call node 8 is labeled '(`D`, `MA`, `GY`)'; similarly, the data-dependence edge from the actual-out node for `*p` at that call node to node 11 is labeled '(`MA`, `D`, `GY`)'.

Because the SDG can contain placeholder definitions and uses at parameter nodes, the data-dependence edges that are incident from, or incident to, such nodes have placeholder definition and use types associated with them; such definitions or uses are always direct. In Figure 10, such data-dependence edges—whose source contains a placeholder definition type or whose target contains a placeholder use type—are distinguished. For example, the data-dependence edge from the formal-in node for `sum` at call node 8 to node 13 has a placeholder definition type associated with it.

Second, like the annotation on data-dependence edges, we annotate each summary edge with the types of data-dependence to which that edge corresponds to. Because data-dependence edges have types associated with them, the summary edges computed using those data dependences also have types associated with them—these types are the types of data dependences that are followed while computing the summary edges. For example, the SDG in Figure 10 contains the summary edges that are created by traversing only data-dependence type (`D`, `D`, `G`); thus, the summary edges are labeled '(`D`, `D`, `G`)' in the figure.

Extensions to the SDG-based slicing algorithm. To compute a slice for criterion $< s, V, T >$, the SDG must contain summary edges for data-dependence types T. After the summary edges are computed, the slicing algorithm proceeds like the two-phase slicing algorithm [22]. During the first phase, the algorithm traverses backward along control-dependence, data-dependence, call, parameter-in, and summary edges. During the second phase, the algorithm traverses backward along control-dependence, data-dependence, parameter-out, and summary edges. We extend the algorithm in two ways.

First, the algorithm traverses backward along a data-dependence or a summary edge only if the data-dependence types associated with that edge appear in the set T of data-dependence types specified in the slicing criterion.

Second, we extend the algorithm to accommodate placeholder definition and use types, which, as mentioned earlier, occur at formal-in and formal-out nodes, respectively. The SDG represents interprocedural data dependences by creating parameter nodes at procedure boundaries and connecting them using

parameter-in and parameter-out edges. Each interprocedural data dependence occurs in the SDG as a path in which each node except the first and the last is a parameter node. For example, let $(d, n_{a_{in}}, n_{f_{in}}, u)$ be such a path in the SDG that represents an interprocedural data dependence (d, u, v). In the path, d is the node that contains the actual definition, $n_{a_{in}}$ is an actual-in node, $n_{f_{in}}$ is a formal-in node, and u is the node that contains the actual use. The path contains two data-dependence edges, $(d, n_{a_{in}})$ and $(n_{f_{in}}, u)$,[10] each of which has an rd type associated with it. These two rd types, $RD_{(d,n_{a_{in}})}$ and $RD_{(n_{f_{in}},u)}$, represent the types of paths over which the definitions reach the respective uses. The rd type for the interprocedural data dependence (d, u, v), $RD_{(d,u)}$, is a composition (explained below) of $RD_{(d,n_{a_{in}})}$ and $RD_{(n_{f_{in}},u)}$.

If, while slicing, the algorithm reaches u, it must eventually include d in the slice provided that the definition type at d, the use type at u, and $RD_{(d,u)}$ match the components of the data-dependence types being sliced for (specified by parameter T in the slicing criterion). However, on reaching u, the slicing algorithm does not have information about d and $RD_{(d,u)}$; it has information only about the use type at u and the rd type $RD_{(n_{f_{in}},u)}$, based on which it must decide whether to visit node $n_{f_{in}}$ and continue slicing from that node. To do this, the algorithm reconstructs $RD_{(d,u)}$ by composing the rd types—for data dependences involving placeholder definitions/uses—that it encounters. At each node, the algorithm saves the composed rd type. If, at any point, the composed rd type is not implied by the rd types specified in the slicing criterion, the algorithm does not traverse the corresponding data-dependence edge.

A composition of two rd types is based on an ordering among the path colors: green < yellow < red. Path color *a subsumes* path color *b* if and only if $a \geq b$. The *composition* operation on two rd types RD_1 and RD_2 performs a pair-wise comparison of the path colors that appear in RD_1 and RD_2 and includes the subsuming color for each pair in the composed RD type. RD_1 *implies* RD_2 if and only if each path color that appears in RD_2 also appears in RD_1.

To illustrate the actions of the algorithm using the above example, on reaching node u, the algorithm extracts $RD_{(n_{f_{in}},u)}$ and composes it with the rd type, if any, saved previously at node u.[11] Next, the algorithm checks whether the composed rd type is implied by the rd types specified in the slicing criterion: if it is, the algorithm traverses edge $(n_{f_{in}}, u)$ and saves the composed rd type at node $n_{f_{in}}$; otherwise, the algorithm does not traverse edge $(n_{f_{in}}, u)$. Next, the algorithm traverses edge $(n_{a_{in}}, n_{f_{in}})$ and copies the saved rd type at $n_{f_{in}}$ to $n_{a_{in}}$. Finally, to determine whether d should be included in the slice, the algorithm composes the saved rd type at $n_{a_{in}}$ with $RD_{(d,n_{a_{in}})}$. If the composed rd type is implied by the rd types specified in the slicing criterion and the definition type at d matches the definition types specified in the slicing criterion, the algorithm includes d in the slice.

The nodes included in the slice for criterion <10, {`sum1`}, {(`D`, `D`, `G`)}> are shown in the lighter shade in Figure 10.

4.2.2 SDG-based slicing algorithm

Figure 11 presents the modified SDG-based slicing algorithm, `ComputeSlice`.

Like Horwitz, Reps, and Binkley's slicing algorithm [22], `ComputeSlice` proceeds in two phases. To identify nodes that are included in the slice in each phase, `ComputeSlice` calls function `GetReachableNodes()` (lines 3, 4). During the first call, `GetReachableNodes()` computes reachability starting at the slicing cri-

[10] Edge $(n_{a_{in}}, n_{f_{in}})$ is a parameter-in edge.
[11] If no rd type is saved at u, the composed rd type is simply $RD_{(n_{f_{in}},u)}$.

```
algorithm ComputeSlice
input   < s, V, T >   slicing criterion
output  slice         slice for < s, V, T >
global  G             SDG for program P without summary edges
begin ComputeSlice
1.  compute summary edges for data-dependence types in T
2.  n = node in G corresponding to s
3.  slice = GetReachableNodes( {n}, T, {call, param-in} )
4.  slice = GetReachableNodes( slice, T, {param-out} )
5.  return slice
end ComputeSlice

function GetReachableNodes
input   N               set of SDG nodes from which to start traversal
        T               set of data-dependence types
        interEdgeTypes  interprocedural edge types to follow during traversal
output  slice           nodes included in the slice
declare worklist        nodes traversed in the SDG while computing slice
begin GetReachableNodes
6.  worklist = N; slice = N
7.  while worklist ≠ φ do
8.      remove node n from worklist
9.      foreach edge e = (m, n) do
10.         case type of e ∈ {interEdgeTypes ∪ control-dependence}
11.             add m to worklist and slice
12.         case type of e is summary
13.             if types(e) appear in T then
14.                 add m to worklist and slice
15.             endif
16.         case type of e is data-dependence
17.             TraverseDataDepEdge( e, T, worklist, slice )
18.         endcase
19.     endfor
20. endwhile
21. return slice
end GetReachableNodes
```

Figure 11: Algorithm for computing a slice based on data-dependence types. Function `TraverseDataDepEdge()` is shown in Figure 12.

terion. During the second call, `GetReachableNodes()` computes reachability starting at the nodes visited during the first call. Before calling `GetReachableNodes()`, `ComputeSlice` first creates the summary edges corresponding to the data-dependence types specified in T (line 1).

The algorithm for computing summary edges is very similar to the reachability-based algorithm for computing summary edges presented by Reps and colleagues [42]. That algorithm extends paths backwards along data-dependence and control-dependence edges, starting at all formal-out nodes. The algorithm iteratively determines whether formal-in nodes are reachable from formal-out nodes and computes summary edges. Our only modification to that algorithm is that it traverses only those data-dependence edges whose types match the types specified in the slicing criterion.

`GetReachableNodes()` uses a worklist to traverse backward along matching data-dependence edges, control-dependence edges, and the specified types of interprocedural edges—call and parameter-in edges during the first phase, and parameter-out edges during the second phase.

```
function TraverseDataDepEdge
input  e = (m, n)  a data-dependence edge in G
       T           set of data-dependence types
       worklist    nodes traversed in the SDG
       slice       nodes included in the slice
begin TraverseDataDepEdge
 1.  if m not a parameter node and n is not a parameter node then
 2.      if types(e) appear in T then
 3.          add m to worklist and slice
 4.      endif
 5.      return
 6.  endif
 7.  if m is not a parameter node then
 8.      if definition type associated with e does not match definition types in T then
 9.          return
10.      endif
11.  endif
12.  if n is not a parameter node then
13.      if use type associated with e does not match use types in T then
14.          return
15.      endif
16.  endif
17.  extract rd type associated with e
18.  compose edge rd type with rd type stored at node n
19.  if composed rd type is implied by the rd types in T then
20.      store composed rd type at m
21.      add m to worklist and slice
22.  endif
end TraverseDataDepEdge
```

Figure 12: Algorithm traversing a data-dependence edge.

On each iteration through the worklist, the algorithm removes a node n from the worklist (line 8) and processes each edge e incident to n (lines 9–19). If e is a control-dependence edge or a relevant interprocedural edge, the algorithm adds m—the source of the edge—to the worklist and the slice (lines 10–11). If e is a summary edge and the data-dependence types associated with e appear in T, the algorithm adds adds m to the worklist and the slice (lines 12–15). If e is a data-dependence edge, the algorithm calls function `TraverseDataDepEdge()` to process the edge (lines 16–18).

`TraverseDataDepEdge()`, shown in Figure 12, takes four inputs: (1) the edge e to be traversed, (2) the set T of data-dependence types specified in the slicing criterion, (3) the worklist, and (4) the slice. If neither the source nor the sink of e is a parameter node (line 1), the data dependence represented by e involves no placeholder definition or use types. Therefore, the algorithm adds the source of e to the worklist and slice if the data-dependence type associated with e appear in T (lines 2–4). Next, the algorithm checks whether the source of e is a non-parameter node (line 7). If it is, the definition at the source node is an actual definition (and not a placeholder definition); therefore, the type of that definition must be checked with the definition types in T. If the definition type associated with e does not appear in the definition types for the data-dependence types in T, `TraverseDataDepEdge()` returns without adding m to the slice (lines 8–10). Similarly, the algorithm ensures that, if the sink node of e is a non-parameter node, the use type associated with e matches the use types for the data-dependence types in T (lines 12–16). Next, the algorithm extracts the rd type associated with e (line 17), composes it with the rd type stored at the sink of e (line 18), and

checks whether the composed rd type is implied by the rd types for the data-dependence types in T (line 19). If the composed rd type is implied by the rd types in T, the algorithm stores the composed rd type at the source node m (line 20) and adds m to the worklist and the slice (line 21).

The pseudocode in Figures 11 and 12 omits such details of the algorithm as checking whether a node has been visited before adding it to the worklist. For a parameter node n, the algorithm needs to check not only whether n has been visited, but whether n has been visited previously with the rd type computed at n during the current visit. Thus, like the traditional SDG-based slicing algorithm, our slicing algorithm visits each non-parameter node at most once during a slice computation. However, unlike the traditional algorithm, our algorithm can visit a parameter node multiple types—once for each unique rd type—during the computation of a slice. Because there are six rd types, our algorithm can visit a parameter node up to six times.

However, the asymptotic time complexity of our algorithm remains linear in the size of the SDG. Operations, such as matching data-dependence types and composing rd types, can be accomplished in constant time using bit-vector representations.

The algorithm as presented in Figure 11 computes summary edges for each slicing request. An alternative approach can be to precompute summary edges by considering all data-dependence types (i.e., as is done in traditional SDG-based slicing [42]) and annotate each summary edge with each set of data-dependence types to which the summary edge corresponds. In general, a summary edge at call site corresponds to a set of paths from a formal-in node to a formal-out node in the PDG of the called procedure. Each such path can have a distinct set of data-dependence types associated with it. Depending on the number of unique combinations of data-dependence types that occur in the paths, the alternative approach can require exponential space and, therefore, may not be practical.

4.2.3 Incremental slicing technique

Using this new slicing paradigm, we define an incremental slicing technique. The *incremental slicing technique* computes a slice in multiple steps by incorporating additional types of data dependences at each step; the technique thus increases the extent of a slice in an incremental manner. In a typical usage scenario, developers can use the technique to consider stronger types of data dependences first and compute a slice based on those data dependences. Then, they can use the technique to augment the slice by considering additional, weaker data dependences and adding to the slice statements that affect the criterion through the weaker data dependences. Alternatively, developers may start by computing a slice based on weaker data dependences and later augment the slice by considering stronger data dependences.

For example, the nodes shown in the lighter shade in Figure 10 are included in the slice for $< 10, \{\texttt{sum1}\}, \{(\texttt{D}, \texttt{D}, \texttt{G})\} >$. Using the incremental technique, when data-dependence types (`D`, `D`, `GY`) and (`D`, `D`, `GR`) are also considered, the nodes shown in the darker shade are added to the slice.

4.2.4 Empirical results

To investigate the incremental slicing technique in practice, we performed an empirical evaluation using our C subjects. We implemented the modified SDG-construction algorithm and the modified SDG-based slicing algorithm using the ARISTOTLE analysis system [5]. Our implementation takes as input a slicing criterion consisting of the SDG node to start the slicing and the set of data-dependence types to traverse

Table 9: The 54 types of data dependences.

RD type	(D, D)	(D, SA)	(D, MA)	(SA, D)	(SA, SA)	(SA, MA)	(MA, D)	(MA, SA)	(MA, MA)
G	t1	t7	t13	t19	t25	t31	t37	t43	t49
GY	t2	t8	t14	t20	t26	t32	t38	t44	t50
GR	t3	t9	t15	t21	t27	t33	t39	t45	t51
GYR	t4	t10	t16	t22	t28	t34	t40	t46	t52
Y	t5	t11	t17	t23	t29	t35	t41	t47	t53
YR	t6	t12	t18	t24	t30	t36	t42	t48	t54

while computing the slice. Then, it computes the summary edges required for the specified data-dependence types. Finally, it traverses the SDG, starting at the criterion and following only the specified types of data dependences, and computes the set of nodes reachable from the criterion.

Goals and method. The overall goal of the empirical evaluation was to investigate whether incremental slicing can be useful in assisting software-engineering tasks. In particular, we wanted to evaluate the usefulness of incremental slicing for debugging.

First, we investigated how incremental approximate dynamic slices can be used to narrow the search space during fault detection and potentially reduce the cost of debugging. An *approximate dynamic slice* is an imprecise approximation of the true dynamic slice—it is computed by intersecting the statements in a static slice with the set of statements that are executed by a test input. In some cases, an approximate dynamic slice can contain unnecessary statements, but, in general, it provides a good approximation of the true dynamic slice and is much less expensive to compute [2]. Dynamic slices are more appropriate for applications such as debugging. Unlike static slices, which include all dependences that could occur in any execution of a program, dynamic slices include only those dependences that occur during a particular execution of a program; for debugging, the relevant execution is a fault-revealing execution. Thus, dynamic slices exclude all statements that, although related to the slicing criterion through chains of data or control dependences, are irrelevant during a fault-revealing execution of the program.

For the first study, we used the subject `space`, for which we have several versions with known faults and several fault-revealing test inputs for each version. We selected 15 versions of `space`, each with a known fault. For each version, we selected a fault-revealing test input and a slicing criterion at an appropriate output statement of the version. Next, we examined the distribution of data-dependence types for `space` and, based on the occurrences of various types, selected nine combinations of data-dependence types for computing the slices: {t1}, {t1–t2}, {t1–t3}, {t1–t5}, {t1–t19}, {t1–t23}, {t1–t25}, {t1–t26}, and {t1–t54}. We use the names t1, t2, ... , t54 to refer succinctly to the 54 types of data dependences; Table 9 maps these names to the types to which they correspond. Using these types, for each version, we computed incremental static slices and intersected them with the statement trace of the fault-revealing test input to obtain incremental approximate dynamic slices for the version. We then examined the increments for the occurrence of the fault.

Second, we evaluated whether the results of incremental slicing generalize to additional subjects. To do this, we examined how the sizes of static slices increase as additional types of data dependences are considered during the computation of the slices. For this study, we used the 13 C subjects listed in Table 3. For each subject, we determined, based on the distribution of data-dependence types, the appropriate incremental

Table 10: Sets of slices computed for each subject. For each increment, we computed the slices starting at five randomly selected nodes in the PDG of each procedure in the program.

Subject	Number of incremental slices	Types of data dependences included in the incremental slices
armenu	3	S1{t1} S2{t1–t3} S3{t1–t54}
bison	3	S1{t1} S2{t1–t3} S3{t1–t54}
dejavu	3	S1{t1} S2{t1–t3} S3{t1–t54}
flex	3	S1{t1} S2{t1–t3} S3{t1–t54}
larn	5	S1{t1} S2{t1–t2} S3{t1–t5} S4{t1–t20} S5{t1–t54}
lharc	3	S1{t1} S2{t1–t3} S3{t1–t54}
mpegplay	4	S1{t1} S2{t1–t3} S3{t1–t26} S4{t1–t54}
mpegplayer	4	S1{t1} S2{t1–t3} S3{t1–t29} S4{t1–t54}
sed	4	S1{t1} S2{t1–t3} S3{t1–t26} S4{t1–t54}
space	4	S1{t1} S2{t1–t3} S3{t1–t5} S4{t1–t54}
T-W-MC	4	S1{t1} S2{t1–t3} S3{t1–t26} S4{t1–t54}
unzip	3	S1{t1} S2{t1–t3} S3{t1–t54}
xearth	4	S1{t1} S2{t1–t3} S3{t1–t26} S4{t1–t54}

Table 11: Sizes of incremental approximate dynamic slices for 15 versions of `space` for fault detection.

	Inc. 1		Inc. 2		Inc. 3		Inc. 4		Inc. 5		Inc. 6		Inc. 7		Inc. 8		Inc. 9	
Ver	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size
1	38	38	0	38	126	164	2	166	**64**	**230**	50	280	12	292	12	304	1	305
2	181	181	0	181	163	163	7	351	**222**	**573**	73	646	82	728	22	750	1	751
3	172	172	0	0	161	161	7	340	130	470	**89**	**559**	85	644	26	26	1	671
4	179	179	0	179	159	338	7	345	137	482	**95**	**577**	78	655	26	681	1	682
5	195	195	0	195	162	357	7	364	164	164	**127**	**655**	114	769	28	797	1	798
6	187	187	0	187	163	350	7	357	**170**	**527**	128	655	74	729	29	758	1	759
7	205	205	0	205	187	392	7	399	204	603	**81**	**684**	98	782	24	806	1	807
8	187	187	0	187	163	350	7	357	**164**	**521**	123	644	78	722	26	748	1	749
9	188	188	0	188	157	345	7	352	182	534	**86**	**620**	110	730	23	753	1	754
10	**189**	**189**	0	189	161	350	7	357	170	527	64	591	93	684	27	711	1	712
11	56	56	0	56	128	184	2	186	**76**	**262**	96	358	47	405	40	445	1	446
12	173	173	0	173	**152**	**325**	6	331	131	462	62	524	82	606	24	630	1	631
13	94	94	0	94	**127**	**221**	4	225	95	320	58	378	82	460	19	479	1	480
14	38	38	0	38	126	164	2	166	**61**	**227**	32	259	12	271	12	283	1	284
15	**185**	**185**	0	185	150	335	7	342	149	491	87	578	98	676	28	704	1	705

slices to compute. We selected the increments such that each increment included at least an additional 5% of the data dependences in the program. Table 10 shows the number of slice increments that were computed for each subject; it also shows, for each slice increment, the data-dependence types that were traversed while computing the slices. For example, consider the entry for `bison` in Table 10. We computed three sets of slices for `bison`: S1, S2, and S3. The slices in the first increment were based only on data-dependence type t1, whereas those in the second and the third increments were based on data-dependence types t1 through t3 and t1 through t54, respectively. For each slice increment, we selected five slicing criteria at random and computed a slice for each of those criteria. We then examined the differences in the sizes of the increments.

Results and analysis. Table 11 presents the results of the first study. The table shows, for each of the 15 versions of `space` that we used, the sizes of the nine incremental slices: the table shows the cumulative size for each increment and the increase in the slice size at each increment. For example, for version 1, the first increment contained 38 statements, the second increment included no additional statements; the third increments contained 126 statements in addition to the 38 from the previous increment.

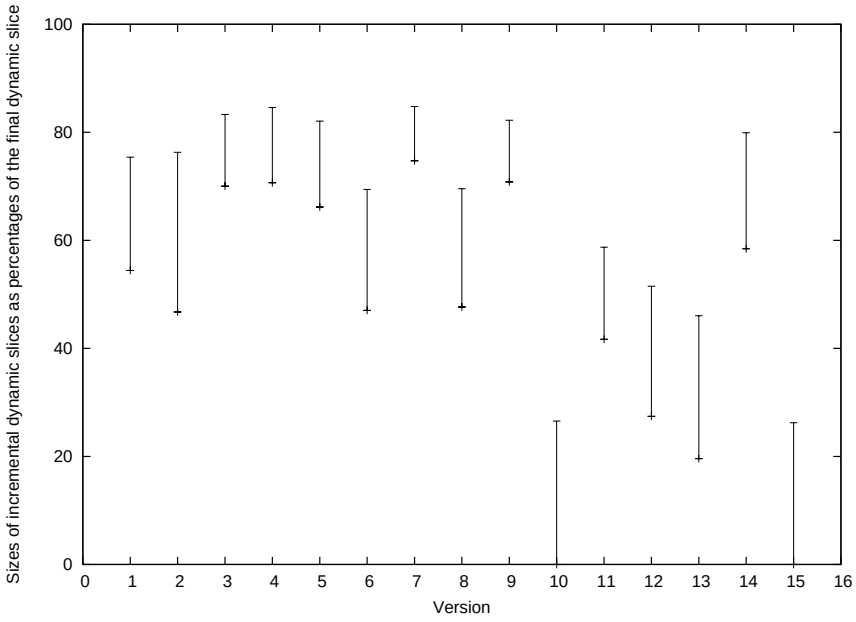

Figure 13: Size of the dynamic slice increment that contained the fault, for each of the 15 versions of `space`.

For each version, increment 9 is the approximate dynamic slice that is computed by including all data dependences. In the alternative debugging scenario, in which testers do not compute incremental slices based on data-dependence types, they would compute increment 9 in one step and then try to locate the fault among the statements included in the slice. The number of statements in the last increment, and thus the number of statements that testers would have to examine, varies from 284, for version 14, to 807, for version 7. However, using incremental slicing, the tester would compute the approximate dynamic slice in increments and examine only the additional statements included in each increment to identify the fault. The increments shown in bold are the first ones that contain the fault for each version.[12] For example, for version 13, the tester would examine 94 statements in the first increment, none in the second, and then 127 in the third to locate the fault; the tester need not examine the remaining 259 statements that appear in the remaining slice increments and that may have to be examined in the alternative debugging scenario.

Using incremental slicing can potentially help speed up the process of locating a fault if the fault occurs in an increment computed before the last increment. Thus, the testers can avoid examining the statements that would appear only in the successive increments. In the worst case, the fault might appear in the last increment (a case that never occurs in our study); in this case, the testers would have to examine as many statements as they would in the alternative debugging scenario. Moreover, incremental slicing allows the testers to examine a smaller set of potentially fault-containing statements at a time and, in doing so, can make the task of locating the fault easier.

The data in Table 11 show that, for 11 of the 15 versions, the fault first appears in either the fifth or the sixth increment. For two versions, the fault appears in the third increment and, for another two, in the first increment. The fault never occurs in the last increment, which indicates that for these versions, using incremental slicing can reduce the number of statements that need to be examined for faults.

[12]Because each successive increment includes all statements from the previous increments, all increments subsequent to the ones shown in bold also contain the faults.

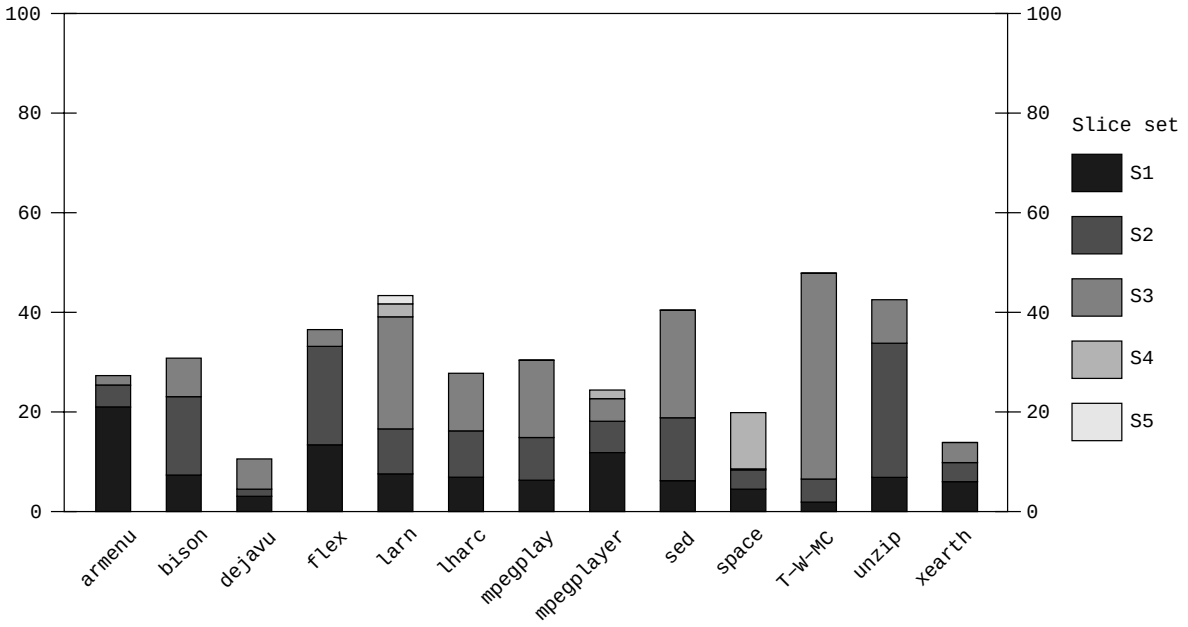

Figure 14: Increase in the sizes of the slices for the slice increments listed in Table 10. For each subject, the segmented bar illustrates the average increase in the slice size from one increment to the next.

Figure 13 shows, for each version, the size of the dynamic slice increment that contained the fault; it shows the size as a percentage of the size of the last incremental slice. The horizontal axis lists the 15 versions of `space`. The length of each line in the figure represents the percentage of additional statements—over previous increments—that were included in the increment containing the fault. The top of each line represents the total percentage of statements that would be examined, including those that appear in the fault-containing increment; the bottom of each line represents the percentage of statements that are examined prior to examining the fault-containing increment. The percentage of statements in the complete incremental slice that need not be examined is at least 15% in each version, and is as high as 74% for two versions. The sizes of the increments that contain the fault vary from 10% to 30% of the final dynamic slice.

Figure 14 presents the results of the second study, in which we computed the slices listed in Table 10. The vertical axis represents the sizes of the slices as percentages of the number of statements in the program. Each segmented bar in Figure 14 illustrates the average increase in the slice sizes for an increment over the previous increment. For example, consider the segmented bar for `bison`. The average size of the slices in set S1, which were computed for data-dependence type t1, is 7% of the program size. The slices in set S2 were computed for data-dependence types t1 through t3. On average, the slices in S2 are larger than the slices in S1 by 16% of the program statements; therefore, the average size of the slices in S2 is 23% of the program size. Similarly, the slices in set S3, which were computed using all types of data dependences, include on average an additional 8% of the program statements; the average size of the slices in S3 is thus 31% of the program size.

For some subjects, a subsequent increment caused a negligible increase in the size of the slice from the previous increment; the segments corresponding to such increments are not discernible in Figure 14. For example, the last increment for `mpegplay` increased the sizes of the slices from the previous increment by less

than 0.1% of the program size. Similarly, the last increment for `sed` also caused the slices to grow marginally. In one of the slice increments—the last increment for `xearth`—none of slices from the previous increment showed an increase in size.

The increase in the size of the slices varies across the subjects as additional data-dependence types are considered. For example, on average, the slice sizes for `armenu` increase by 4% of the program size when data-dependence types t2 and t3 are considered in addition to data-dependence type t1. However, for `bison`, the inclusion of those types causes the slice sizes to increase by 16%.

Overall, the data show that few slice increments cause the slices to increase in size. For some increments, the increase is marginal, whereas, for others, it is substantial. However, limiting the types of data dependences that are traversed during slicing can cause the slices to be smaller, and thus, more amenable for accomplishing the task for which the slices are computed.

Discussion. Our empirical studies indicate that incremental slicing can be effective in computing a complete slice in multiple steps. Each step increases the size of the slice by traversing additional types of data dependences. The technique provides a systematic way of reducing the size of a slice, by considering only those types of data dependences that are of interest. When applied to fault detection, the technique lets the testers focus on smaller subsets of the fault space—the set of statements that potentially contain the fault. Instead of having to search the entire fault space, the technique lets the testers partition the fault space and examine the partitions separately. Thus, as the results of our first study show, the testers need not examine the entire fault space, which can reduce the fault-detection time. The second study shows that the results of incremental slicing generalize to more subjects than the one used in the first study, thus making the technique more generally applicable.

5 Related Work

Throughout this section, we use our color-based terminology to discuss the classifications provided by other authors, even though none of those authors actually use colors in their work. We do this for ease of comparison with our classification.

Ostrand and Weyuker [38] extend the traditional data-flow testing techniques [15, 41] to programs that contain pointers and aliasing. To define testing criteria that adequately test the data-flow relationships in programs with pointers, they consider the effects of pointers and aliasing on definitions and uses. They classify data dependences based on types of definitions, uses, and paths between definitions and uses. They identify two types of definitions and uses: definite and possible—they do not distinguish single-alias accesses and, instead, group single-alias accesses with definite accesses. They distinguish three types of paths, based on the occurrences of no yellow paths, some yellow paths, and all yellow paths, between definitions and uses. Based on these types, they define four types of data dependences. A *strong data dependence* involves a definite definition, a definite use, and no yellow paths between the definition and use. A *firm data dependence* involves a definite definition, a definite use, and at least one green and one yellow path from the definition to the use. A *weak data dependence* involves a definite definition, a definite use, and all yellow paths between the definition and use. A *very weak data dependence* involves either a possible definition or a possible use. Ostrand and Weyuker define new data-flow testing criteria designed to cover the four types of data

dependences. They also discuss how their classification can be used to prioritize the coverage of certain types of data dependences over others, to meet the constraints of limited time and resources.

Ostrand and Weyuker's classification is much coarser grained than ours. In their classification, a definite definition (or use) includes both direct and single-alias definitions (or uses). They identify three types of paths—they do not distinguish the occurrence of red paths, like we do. Their classification of paths is not directly comparable with ours because certain yellow paths in our classification—those in which the redefinition occurs through a single-alias access—are classified as red paths in their classification.

Apart from testing, Ostrand and Weyuker do not investigate other applications of data-dependence classification. Also, they do not consider classification based on spans.

Merlo and Antoniol [31, 32] present techniques to identify implications between nodes and data dependences in the presence of pointers. They distinguish definite and possible definitions and uses and, based on these, identify definite and possible data dependences. A *definite data dependence* involves a definite definition, a definite use, and at least one green path between the definition and the use. A *possible data dependence* involves either a possible definition or a possible use, and at least one green or yellow path between the definition and the use. Merlo and Antoniol do not mention whether they consider a single-alias access to be a definite access.

The goal of Merlo and Antoniol's work is to identify, through static analysis, implications between nodes and data dependences. They define relations, based on dominance, to compute, for each node, the set of data dependences whose coverage is implied by the coverage of that node. The application of our classification provides an alternative way to estimate the data-flow coverage achieved by a statement-adequate test suite. Moreover, unlike their approach, our approach is applicable to interprocedural data dependences. However, our approach is more limited than theirs in two ways. First, the application of our classification provides a lower bound on the number of data dependences whose coverage can be inferred from statement coverage. Our approach considers only those data dependences in which all paths between the definition and the use are green and, in addition, either the definition dominates the use or the use postdominates the definition. In general, it is possible to infer the coverage of a data dependence from the coverage of a node, even if some paths between the definition and use contain kills or there is no dominance/postdominance relation between the definition and the use. Second, our approach is not applicable in cases in which 100% statement coverage cannot be assumed for a program. However, our goal in this work is not to describe a general, comprehensive approach for inferring coverage of data dependences from coverage of statements; instead, inferring data-flow coverage, albeit conservatively, is an application and benefit of our classification.

Marré and Bertolino [30] define subsumption relations among data dependences to determine the data dependences whose coverage can be inferred from the coverage of other data dependences. Their approach identifies a *spanning set* of data dependences, which is a minimal set of data dependences whose coverage ensures the coverage of all data dependences in the program. Marré and Bertolino do not consider the effects of pointer dereferences on identifying the spanning set of data dependences.

Pande, Landi, and Ryder [39] describe an algorithm for computing interprocedural reaching definitions in the presence of pointers. To compute interprocedural reaching definition, they describe conditional reaching definitions. A *conditional reaching definition* is a reaching definition that reaches node n under assumed alias relations and reaching definitions at the entry of the CFG to which n belongs.

Several researchers have considered the effects of pointers on program slicing and have presented results

to perform slicing more effectively in the presence of pointers (e.g. [1, 6, 9, 10, 28]). Some researchers have also evaluated the effects of the precision of the pointer analysis on subsequent analyses, such as the computation of def-use associations (e.g., [47, 48]) and program slicing (e.g., [8, 27, 44]). However, none of that research distinguishes data dependences based on types of the definition, the use, and the paths between the definition and the use.

Other researchers (e.g. [11, 17]) have investigated various ways to reduce the sizes of slices. However, they have not considered classifying data dependences and computing slices based on different types of data dependences as a means of reducing the sizes of slices.

6 Summary and Future Work

In this paper, we presented two techniques for classifying data dependences in programs that use pointers. The first technique classifies a data dependence based on the type of definition, the type of use, and the types of paths between the definition and the use. The technique classifies definitions and uses into three types based on the occurrences of pointer dereferences; it classifies paths between definitions and uses into six types based on the occurrences of definite, possible, or no redefinitions of the relevant variables along the paths. Using this classification technique, data dependences can be classified into 54 types. The second technique classifies data dependences based on their spans—it measures the extent or the reach of a data dependence in a program and can be computed at the procedure level and at the statement level. Although our techniques are intended to classify data dependences in the presence of pointer dereferences, they are also applicable to programs that do not contain pointer dereferences.

We implemented the two classification techniques and investigated the occurrences of data dependences in practice, for a set of C programs. The results of the studies indicated that, for our subjects, the five most-frequently-occurring data-dependence types can account for as much as 90% of the data dependences in the programs (Figure 5). Data about the most-frequently-occurring types in a program (Table 5) can be used to characterize programs based on the complexity of their data dependences, as confirmed through manual inspection of the programs. Information about data-dependence spans can also be used to characterize programs.

We presented two applications of the classification techniques: data-flow testing and program slicing. In the first application, we explored different ways in which the classification can be used to facilitate data-flow testing. We used the classification to determine the data-flow coverage achieved through less-expensive testing such as statement or branch testing. We also used the classification to order data dependences for coverage and to aid in generating test inputs for covering them. Finally, we used the classification to determine the appropriate verification technique for different data dependences—some data dependences may be suitable for verification through testing, whereas, for others, because of their complexity, alternative verification techniques, such as inspections, may be more appropriate. In our empirical evaluation, we showed a possible application of our approach to a set of C subjects. The results of the study indicate that the approach is practical and effective; for each subject, we were able to identify the subset of data dependences covered by statement coverage and to suggest a verification technique for the remaining data dependences, based on the expected complexity of covering them.

In the second application, we presented a new slicing paradigm in which slices are computed by traversing

data dependences selectively, based on their types. The new slicing paradigm can be used to compute a slice incrementally. The incremental slicing technique computes a complete slice in multiple steps by traversing additional types of data dependences during each successive step. We performed an empirical study to investigate how incremental approximate dynamic slices can be used to reduce the fault-detection time. Our results indicated that, using incremental slicing, testers need not examine the entire fault space (Table 11, Figure 13). Moreover, testers can examine a much smaller fault space at a time, which can speed up the process of locating a fault.

We performed a second study to evaluate whether the results of incremental slicing generalize to more subjects. We investigated the increase in the sizes of static slices as additional types of data dependences are considered for a set of C subjects. The outcome of the study shows that incremental slicing's results are consistent across the subjects considered (Figure 14), thus making the technique more generally applicable for tasks such as program comprehension and debugging.

In our work, we have not examined the dynamic occurrences of different types of data dependences. A promising area of future work would be to use dynamic analysis to investigate whether the type of a data dependence can be used to predict the feasibility of that data dependence. Another potential area of future work is to apply the classification to other languages, notably Java. The classification techniques may need to be extended or modified to accommodate unique features of Java. The patterns in the occurrences of data dependences that we have observed for C programs would likely differ for Java programs. An additional direction for future work is to design and implement human studies to assess and refine our approach for data-flow testing. Yet another possible future direction is to investigate the use of our classification techniques to study the coupling between different modules in a program. We expect that computing coupling measures based on the types of data dependences between two modules can provide a better understanding of the actual coupling between such modules. Finally, we are interested in exploring visualization techniques for presenting, in an intuitive way, information about the data dependences within a program and their types (e.g., by letting the user visualize only a subset of data dependences and navigate between definitions, uses, and parts of the program in the spans).

Acknowledgments

Gregg Rothermel provided useful suggestions and comments that helped improve the paper. Donglin Liang provided an implementation of his parametric alias analysis and helped with its installation. This work was supported in part by a grant from Boeing Commercial Airplanes to Georgia Tech, by National Science Foundation awards CCR-9988294, CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 60–73, October 1991.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

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

[4] L. O. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.

[5] Aristotle Research Group. ARISTOTLE: Software engineering tools. http://www.cc.gatech.edu/aristotle/, 2000.

[6] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of ACM SIGSOFT 6^{th} International Symposium on the Foundations of Software Engineering*, pages 46–55, November 1998.

[7] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[8] L. Bent, D. C. Atkinson, and W. G. Griswold. A comparative study of two whole-program slicers for C. Technical Report UCSD TR CS2000-0643, University of California at San Diego, May 2000.

[9] D. W. Binkley. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum*, pages 261–268, November 1993.

[10] D. W. Binkley and J. R. Lyle. Application of the pointer state subgraph to static program slicing. *The Journal of Systems and Software*, 40(1):17–27, January 1998.

[11] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998.

[12] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[14] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[15] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[16] P. G. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[17] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Proceedings of the 5^{th} International Workshop on Program Comprehension*, pages 70–79, March 1997.

[18] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20^{th} International Conference on Software Engineering*, pages 74–83, April 1998.

[19] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, March 1997.

[20] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[21] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[23] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[24] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[25] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[26] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[27] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of ESEC/FSE '99 7^{th} European Software Engineering Conference and 7^{th} ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 199–215. Springer-Verlag, September 1999.

[28] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the International Conference on Software Maintenance*, pages 421–432, August–September 1999.

[29] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of 2001 Static Analysis Symposium*, pages 279–298, July 2001.

[30] M. Marré and A. Bertolino. Unconstrained duas and their use in achieving all-uses coverage. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 147–157, January 1996.

[31] E. Merlo and G. Antoniol. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *Proceedings of CASCON '99*, pages 173–186, November 1999.

[32] E. Merlo and G. Antoniol. Pointer sensitive def-use pre-dominance, post-dominance and synchronous dominance relations for unconstrained def-use intraprocedural computation. Technical Report EPM/RT-00/01, Ecole Polytechnique of Montreal, March 2000.

[33] S. Ntafos. On required elements testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[34] S. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

[35] A. Orso, D. Liang, S. Sinha, and M. J. Harrold. A framework for understanding data dependences. Technical Report GIT-CC-02-13, College of Computing, Georgia Institute of Technology, March 2002.

[36] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. In *Proceedings of the 9^{th} International Workshop on Program Comprehension*, pages 39–49, May 2001.

[37] A. Orso, S. Sinha, and M. J. Harrold. Incremental slicing based on data-dependence types. In *Proceedings of the International Conference on Software Maintenance*, pages 158–167, November 2001.

[38] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 74–86, October 1991.

[39] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[40] Programming Language Research Group. PROLANGS Analysis Framework. http://www.prolangs.rutgers.edu/, Rutgers University, 1998.

[41] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[42] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20, December 1994.

[43] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[44] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *4th International Static Analysis Symposium*, volume 1302 of *LNCS*, pages 16–34, September 1997.

[45] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, May 1999.

[46] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[47] P. Tonella. Effects of different flow insensitive points-to analyses on DEF/USE sets. In *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering*, pages 62–69, March 1999.

[48] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis. *Journal of Software Maintenance: Research and Practice*, 11(2):117–142, March–April 1999.

[49] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.