Classifying Data Dependences in the Presence of Pointers for Program Comprehension, Testing, and Debugging

ALESSANDRO ORSO, SAURABH SINHA, and MARY JEAN HARROLD
College of Computing, Georgia Institute of Technology

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 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. 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 approach that computes slices based on types of data dependences. Based on the new approach, 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.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; *Testing tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering* ; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*

General Terms: Algorithms,Experimentation,Measurement

Additional Key Words and Phrases: Data dependences, data-flow testing, debugging, incremental slicing, pointers, program comprehension, program slicing

This work was supported in part by a grant from Boeing Commercial Airplanes to Georgia Tech, by National Science Foundation awards CCR-0306372, CCR-0205422, CCR-9988294, CCR-0209322, and SBE-0123532 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

Authors' address: College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332

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 data dependences among program variables. Such dependences relate statements that assign values to variables to statements that use those values. In the absence of pointers, definitions and uses of variables can be identified by using only syntactic information. However, 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, 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 task 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 to 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*, that is, the likelihood that a data dependence identified statically occurs dynamically. The technique extends the classification presented by Ostrand and Weyuker [1991] 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 an existing algorithm for computing interprocedural data dependences.

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 such 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. In the first application, we investigate how the classification techniques can be used to facilitate data-flow testing. Although data-flow testing techniques [Frankl and Weyuker 1988; Rapps and Weyuker 1985] have long been known, they are rarely used in practice, primarily because of their high costs [Beizer 1990]. The main factors that contribute to the high costs are (1) the large number of data dependences to be covered, a number of which may be infeasible,[1] (2) the

[1] A data dependence is *infeasible* if there exists no input to the program that causes that association to be exercised.

difficulty of generating test inputs to cover the data dependences, and (3) the expensive program instrumentation required to determine the data dependences that are covered by test inputs.

We investigate how classifying data dependences can help reduce the costs of data-flow testing by providing ways 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 a 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.

In the second application, we present a new slicing approach that computes program slices [Weiser 1984] by considering only a subset of data dependences. This approach 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.

Based on the new slicing approach, 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 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 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, 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. For debugging, the approach also lets developers focus on only those data dependences that are related to the failure.

To evaluate our incremental slicing approach, we implemented the technique—by extending the SDG-based approach for slicing [Horwitz et al. 1990; Reps et al. 1994; Sinha et al. 1999]—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 a number of subjects, thus making the technique more likely to be applicable.

The main contributions of the paper are:

—Two techniques, one based on data-dependence types and the other based on data-dependence 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 that demonstrate how the classification can be used to estimate data-flow coverage and select the appropriate verification technique for data dependences.

—A new approach 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 approach in which slices are computed based on data-dependence types; based on the approach, 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.

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`}.

[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, j;
 main() {
   int sum;
1  read i;
2  read j;
3  sum = 0;
4  while ( i < 10 ) {
5    sum = add( sum );
   }
6  print sum;
 }
```

```
 int add( int sum ) {
7  if ( sum > 100 ) {
8    i = 9;
   }
9  sum = sum + j;
10 read j;
11 i = i + 1;
12 return sum;  }
```

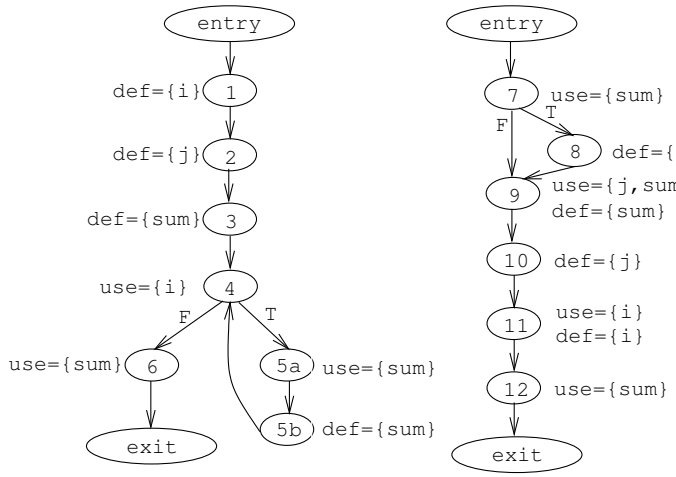

Fig. 1. Program `Sum1` to illustrate definitions, uses, and data dependences (top); control-flow graphs for the program annotated with def and use sets (bottom).

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 $(y, n_1, n_2, \ldots, n_k, z)$ 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 from d_1 to d_2. For example, the definition of `i` at node 11 kills the definition of `i` at node 8.

A *reaching-definition set*, $rd(z)$, defined with respect to a node z, is the set of variable–node pairs $\langle v, y \rangle$ such that $v \in \text{def}(y)$ and there exists a def-clear path with respect to v from y to z. 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 $\langle v, d \rangle \in \text{rd}(u)$. 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 [Aho et al. 1986].

```
    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;
    }
```

Fig. 2. Program Sum2.

2.2 Alias analysis

In languages that contain usage of pointers, the computation of data dependences 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., [Andersen 1994; Landi and Ryder 1992; Liang and Harrold 1999a; Steensgaard 1996]).

2.3 Data-flow testing

Data-flow testing techniques use the data-flow relationships in a program to guide the selection of test inputs (e.g., [Harrold and Soffa 1991; Laski and Korel 1983; Ntafos 1984; Rapps and Weyuker 1985]). For example, the all-defs criterion [Frankl and Weyuker 1988; Rapps and Weyuker 1985] requires the coverage of each definition in the program to some reachable use; the stronger all-uses criterion requires

[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.

the coverage of each data dependence in the program. Other criteria require the coverage of chains (of different lengths) of data dependences [Ntafos 1984].

2.4 Program slicing

A *program slice* of a program $\mathcal{P}$, computed with respect to a *slicing criterion* $\langle s, V \rangle$, 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 [Weiser 1984]. 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) [Horwitz et al. 1990; Sinha et al. 1999] and data-flow equations [Harrold and Ci 1998; Weiser 1984] form two alternative, general classes of slicing techniques. For this work, we extend the SDG-based slicing approach [Horwitz et al. 1990; Reps et al. 1994; Sinha et al. 1999] (the approach based on data-flow equations [Harrold and Ci 1998; Weiser 1984] could be extended similarly). The SDG-based approach precomputes all dependences and represents them in the SDG, and then computes slices using graph reachability.

Unlike static slicing techniques, which consider dependences that can occur in any execution of a program, dynamic slicing techniques [Agrawal and Horgan 1990; Korel and Laski 1988] consider only those dependences that occur in a particular execution of the program; dynamic slicing techniques, thus, ignore those static dependences that do not occur in that execution. A dynamic slicing criterion contains an input to the program, an instance of a program statement in the execution trace, and a set of variables.

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) [Horwitz 1997]. 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 [1991]. 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

[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.

dereference can potentially access one of several memory locations. For example, in program `Sum2` (Figure 2), statement 2 defines `sum2` through direct access. Whereas, 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 statement 8, `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 statement 8. 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 the 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. These types of memory accesses result in either definite or possible definitions and uses. A direct access results in a *definite definition* or *definite use* of the memory location being accessed, whereas a multiple-alias access results in a *possible definition* or *possible use* of the memory location being accessed. A single alias can result in either a definite access or a possible access of a memory location. Consider the two examples shown in Figure 3. The example on the left illustrates the case in which a single-alias access results in a definite access: the use of `a` through `*p` in line 4 is a definite use. The example on the right illustrates the case in which a single-alias access results in a possible access: the definition of `a[i]` in line 4 is a possible definition, and the use of `a[j]` at line 5 is a possible use. (In this case, the definition and the use are not definite because, in general, static analyses cannot distinguish between different elements of an array.) Therefore, to be conservative, we consider a single-alias access to be a possible definition or use of the accessed memory location.

Based on the types of definitions and uses, there are nine possible types of data dependences because both the definition and the use can be of three different types. 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. In the figure, sets *ddef* and *duse* indicate definite definitions and uses, whereas sets *pdef* and *puse* indicate possible definitions and uses.

```
1.  int a, b, *p;
2.  a = 1;
3.  p = &a;
4.  b = *p;
```

```
1.  int i,j;
2.  int a[3] = {1,2,3};
3.  scanf("%d %d", &i, &j);
4.  a[i] = 5;
5.  printf("%d\n", a[j]);
```

Fig. 3. Examples to illustrate single-alias access. In the example on the left, the single-alias use of `a` in line 4 is a definite use, whereas, in the example on the right, the single-alias use of `a[j]` in line 5 is a possible use.

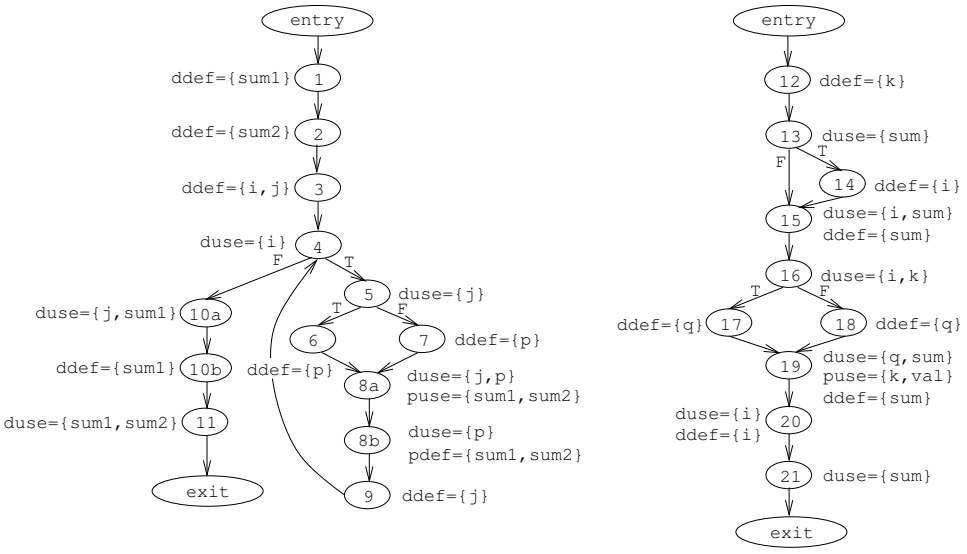

Fig. 4. Control-flow graphs for the procedures in `Sum2` (Figure 2) with definite and possible definition and use sets at each node.

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 information is insufficient—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 would be identified as containing no kills by a conservative data-flow analysis [Aho et al. 1986]. Thus, a definition may actually not reach a use along such a path, although the analysis would not account for this possibility.

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 each path π from d to u into

Table I. 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

one of two types, based on whether the definition at d is killed along 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 $(y, n_1, n_2, \ldots, n_k, z)$ 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 $(y, n_1, n_2, \ldots, n_k, z)$ 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` (Figure 4), 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 $(y, n_1, n_2, \ldots, n_k, z)$ 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 (1, 2, 3, 4, 10a, 10b, 11) is a definite killing path with respect to variable `sum1`, because node 10b contains a definite definition of `sum1`.

To ease the presentation, we associate colors green, yellow, and red with the three types of paths: green (G) with definite def-clear paths, yellow (Y) with possible def-clear paths, and red (R) with definite killing paths. The analogy with a traffic light provides intuition about the meaning of the path colors: 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 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 I lists the 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 meaningful rd types.

Table II. Data dependences, with their types, that occur in program `Sum2`.

Data dependence	Type	Data dependence	Type Type	Data dependence	Type 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`)		

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 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 II 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. Spans also provide an intuitive measure of the complexity of data dependences. A data dependence with smaller span can be understood by examining smaller portions of the program than one with a larger span. The one with larger span extends over a larger portion of a program, with kills or potential kills occuring in several different procedures. Such a data dependence is, thus, likely to be more complex. 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 $\langle proc, occ, color \rangle$; the set contains a triple for the procedure that contains the definition d, a triple for the procedure that contains the use u, and a triple for each procedure that contains a definite or possible kill for the data dependence. Each triple is composed as follows:

—*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`⟩}.

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 $\langle proc, stmt, occ, color \rangle$; 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`⟩}.

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

from 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 and classify data dependences, we use an algorithm previously developed by some of the authors. Reference [Orso et al. 2002] contains a detailed description of the algorithm. Here, we provide only a high-level description of the algorithm and its complexity.

The algorithm extends the algorithm by Harrold and Soffa [1994], which computes interprocedural data dependences in two phases. In the first phase, it 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 this phase, to compute rd types for intraprocedural data dependences, we modify the traditional algorithm for computing reaching definitions; the modified algorithm propagates 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. 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.

To compute spans of interprocedural data dependences, we leverage the same algorithm. 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 definition 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.

The first phase of the algorithm analyzes each procedure separately and the cost of processing a procedure is quadratic in the number of statements in the procedure. In the second phase, the construction of the interprocedural flow graph requires several traversals of each procedure's subgraph, each of which is linear in the size of the subgraph. The number of traversals for a procedure is bounded by the number of non-local variables that are modified by the procedure. The final step of computing interprocedural data dependences requires linear traversals of the interprocedural flow graph, once for each definition (or use) in the program.

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

Table III. Programs used for the empirical studies reported in the paper.

Subject	Description	Procedures	LOC
armenu	ARISTOTLE analysis system user interface	95	6067
bison	Parser generator	131	5542
dejavu	Interprocedural regression test selector	91	3166
flex	Lexical analyzer generator	140	8264
larn	A dungeon-type game program	292	7715
lharc	Compress/extract utility	89	2500
mpegplay	MPEG player	140	12354
mpegplayer	Another MPEG player	106	5380
sed	GNU batch stream editor	77	5418
space	Parser for antenna-array description language	137	6199
T-W-MC	Layout generator for cells in circuit design	225	21379
unzip	Zipfile extract utility	41	2834
xearth	Display program for a shaded image of the earth	101	21333

study. We implemented the reaching-definitions algorithm using the ARISTOTLE analysis system [Harrold and Rothermel 1997]. For alias information, we used the alias analysis described in Reference [Liang and Harrold 2001]; that implementation is based on the PROLANGS Analysis Framework [Programming Language Research Group 1998].

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 III describes the subject programs and lists the number of procedures and the number of non-comment lines of code in each program.

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 overall occurrences of data dependence types across subjects. Finally, we studied the distribution of interprocedural data dependences based on the sizes of their procedure spans.

3.4.2 *Results and analysis.*

Occurrences of data-dependence types within subjects. We begin by examining the number of data dependences and the number of data-dependence types computed for the subject programs. Table IV shows the number of intraprocedural and interprocedural data dependences (DUAs) for each subject. The table also shows the number of data-dependence types that occurred among the intraprocedural and interprocedural data dependences and overall 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 programs 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 interprocedural data dependences than among intraprocedural data dependences.

Table IV. 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

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.

We selected the sets to examine based on the following reasons. Examining top 1 and top 2 was of interest to see in what numbers do the most-frequent and the second most-frequent data-dependence types occur. These numbers tell us whether one or two data-dependence types occur in dominant numbers. (For all the subjects, top 1 and top 2 accounted for more than 50% of the data dependences.) With the top 10 types, we reached close to 100% of the data dependences for each of the subjects; therefore, we did not examine additional types. After top 2, we could have examined linearly top 3, top 4, etc. However, to simplify the presentation of the data, we aggregated the data for those types. We increased the number of types in each aggregate because the number of data dependences of those types were decreasing.

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 IV), few of those types occur in large numbers and

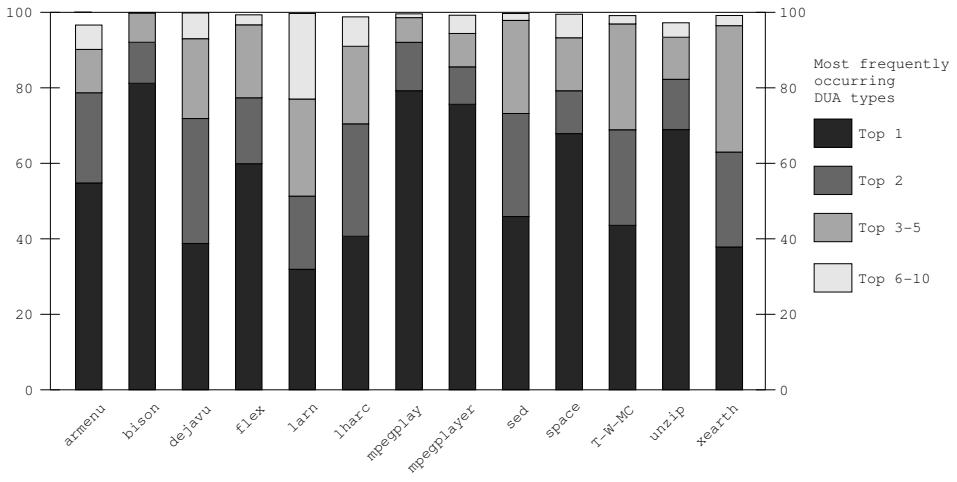

Fig. 5. Percentage of data dependences accounted for by the most-frequently-occurring data-dependence types: top 1, top 2, top 3–5, and top 6–10.

the remaining types occur in very small numbers. For example, although 30 types of data dependences occur in `mpegplay`, 10 of them account for over 99% of the 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 V 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 occurrence of such types in a program indicates that the program manipulates simple data structures and has relatively simple data flow. 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 the uses involve multiple-alias accesses do not appear prominently in Table V. 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 V 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.

Table V. 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)

Table VI. 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

Occurrences of data-dependence types across subjects. Next, we examine, for each data-dependence type, the number of times it occurs over all subjects; Table VI 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 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 need 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

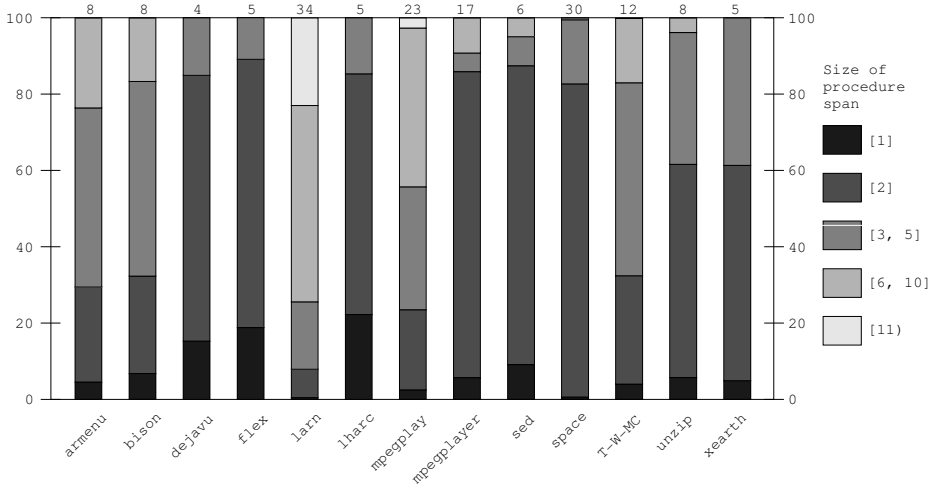

Fig. 6. Distribution of interprocedural data dependences based on the number of elements in the procedure spans. Each segment represents the percentage of interprocedural data dependences that spanned a particular range of procedures. The number at the top of each bar is the size of the largest procedure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

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 greater than five occur in significant numbers in `armenu`, `bison`, `larn`, `mpegplay`, and `T-W-MC`. Spans greater 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 greatest span also occurs in `larn`—it includes 34 of the 179 procedures in that subject. The greatest 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 in our subjects. 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 across our 13 subjects 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. Our results also suggest that examining the most-frequently occurring data-dependence types can help developers infer the

overall complexity of the data-flow relations in a program. 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 programs 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.) However, further empirical evaluation is necessary to determine whether types and spans can be used to characterize programs accurately.

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 VI). Although further empirical evaluation is needed to assess the usefulness of the identified patterns, our initial experience and the results of the study are promising: they suggest that examining such patterns in a program can help the developers to get an overall view of the data-flow structure of a program.

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 information about data-dependence types and spans.

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 [Ostrand and Weyuker 1991]; 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 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 our 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 [Frankl and Weyuker 1988; Rapps and Weyuker 1985]; these techniques provide different 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., coverage of each conditional branch in a program) [Clarke et al. 1989; Ntafos 1988; Rapps and Weyuker 1985]. Previous research has also shown that data-flow testing

can be more effective at detecting faults than branch testing [Frankl and Weiss 1993; Frankl and Weyuker 1993]. However, despite their apparent strengths and benefits, data-flow testing techniques are rarely used in practice, primarily because of their high costs [Beizer 1990]. As mentioned in Section 1, the main factors that contribute to these 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 ordering or prioritizing 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 input 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 test-input generation.* Ostrand and Weyuker [Ostrand and Weyuker 1991] 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 the 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 occur. In contrast, to cover 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 desired 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 data dependence with rd type `G` to be

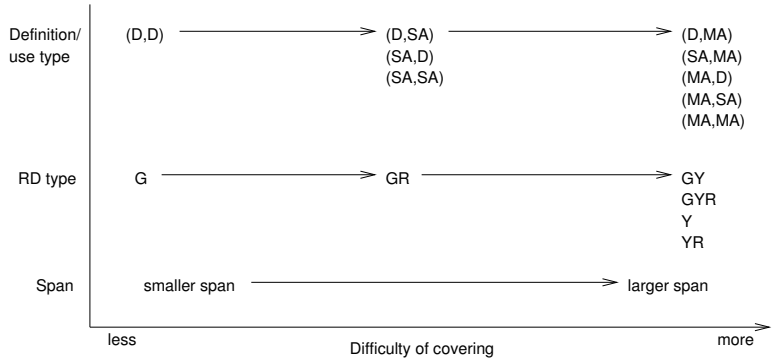

Fig. 7. Relative expected difficulties of covering different types of definitions, uses, rd types, and spans.

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 a given threshold may 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. Alternatively, 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 input 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 [Pande et al. 1994]. 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 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 input, 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 data dependences of type (`D`, `D`, `G`) in which either the definition dominates the use or the use postdominates the definition.[5] Such data dependences 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 in order; we call this criterion *def-use coverage*. Def-use coverage is less expensive than coverage of data dependences 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. 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 inputs 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, alternative verification

[5] 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_j to the end of the program goes through s_i.

techniques, such as code inspection, may be more appropriate. In the absence of additional information about data dependences, such as types and spans, testers have no guidance in determining the appropriate verification technique for different data dependences.

4.1.4 *Empirical results.* To illustrate how the classification of data dependences can be applied, in practice, to data-flow testing, we conducted an empirical 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 empirical 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 likely make test-input generation 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 type `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.[6] 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 type `GR`; for such data dependences, each path from the definition to the use is definite def-clear or definite killing. Thus, intuitively, generating test inputs for such data dependences should be easier than generating inputs 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. We used a threshold of three for the remaining data dependences, whose rd types included a yellow path. Because generating test input in the presence of a yellow path can, in general, be more challenging, we used a smaller threshold value for such data dependences. Table VII lists the criteria that we used in the empirical study to determine the appropriate verification technique.

Note that the values we selected are only 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 input for dependences spanning multiple procedures.

[6]Because data dependences of type (`D`, `D`, `G`) have no kills, their procedure spans contain at most two triples—one for the procedure in which the definition occurs and the other for the procedure in which the use occurs.

Table VII. 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								

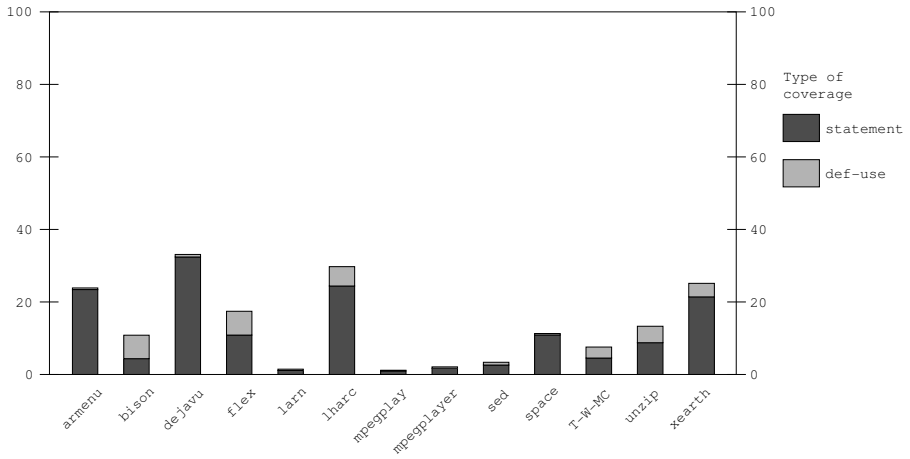

Fig. 8. Percentage of data dependences that are covered by statement coverage and def-use coverage.

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

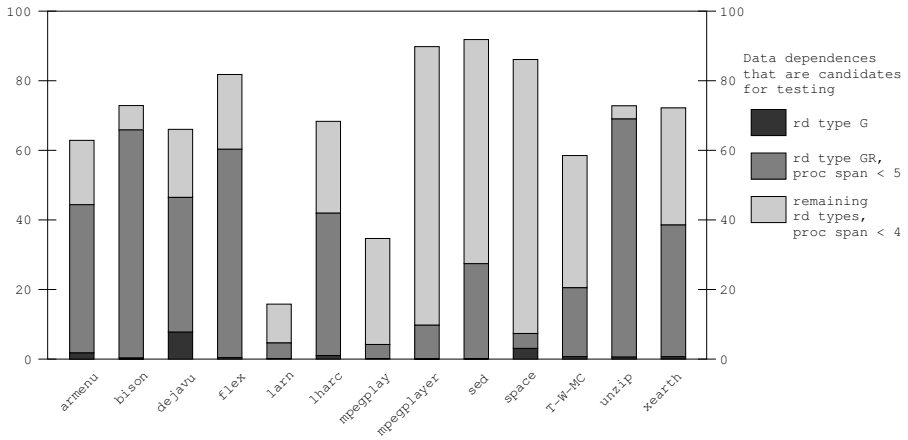

Fig. 9. Percentage of remaining data dependences of different types and procedure spans that can be candidates for data-flow testing.

the definition dominates the use or the use postdominates the definition.[7] 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 indicate that most of the data-dependences of type (`D`, `D`, `G`) occur intraprocedurally,[8] 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 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` (see Row 1 of Table VII). 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

[7]Our analysis tools currently 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.

[8]This follows from the fact that, because we used intraprocedural dominance, the darker segment in Figure 8 represents intraprocedural data dependences of type (`D`,`D`,`G`) in which either the definition dominates the use or the use postdominates the definition. The lighter segment represents the remaining data dependences, including interprocedural ones, of type (`D`,`D`,`G`).

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 four or less. 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 VIII 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 them extremely difficult to be verified through testing. Generating test input for such data dependences, if at all possible, may not be worth the time and effort that it would require. Table VIII 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. Other subjects, such as `dejavu` and `xearth`, have few remaining data dependences and those data dependences span a small percentage of the procedures in those programs.

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 verification is attained,

Table VIII. 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%

the process can be terminated. The results of the study suggest that the approach can be practical and useful. However, further empirical evaluation is needed to determine the effectiveness of ordering data dependences based on types and spans.

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.

4.2 Incremental slicing based on data-dependence types

Traditional slicing techniques (e.g., [Harrold and Ci 1998; Horwitz et al. 1990; Weiser 1984]) 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 approach for slicing, in which the transitive closure is performed over only the specified types of data dependences, rather than over all data dependences. In this slicing approach, a *slicing criterion* is a triple $\langle s, V, T \rangle$, 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 approach using either the SDG-based approach [Horwitz et al. 1990; Reps et al. 1994; Sinha et al. 1999] or the data-flow-based approach [Harrold and Ci 1998; Weiser 1984].

To compute slices in the new approach, using the SDG-based approach, we extend both the SDG and the SDG-based slicing algorithm. We extend the SDG in two ways: (1) we annotate each data-dependence edge with the type of the corre-

sponding data dependence, and (2) we annotate each summary edge with the types of data dependences that are followed while computing the summary edge.[9] Reference [Orso et al. 2003] describes the extensions and illustrates them with examples.

4.2.1 *Incremental slicing technique.* Using this new slicing approach, 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.[10] 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 space constraints, we do not discuss the incremental slicing algorithm; Reference [Orso et al. 2003] presents the details of the algorithm.

4.2.2 *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 [Aristotle Research Group 2000]. 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 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. We compute an *approximate dynamic slice* by intersecting the statements in a static slice with the set of statements that are executed by a test input. In some cases, such an approximate dynamic slice can be *imprecise*—that is, it can contain unnecessary statements—but, in general, it provides a good approximation of the true dynamic slice and is much less expensive to compute [Agrawal and

[9] An SDG contains *data-dependence edges* to represent data dependences and *summary edges* to represent transitive flow of dependences across call sites caused by data dependences, control dependences, or both.

[10] The idea of incremental slicing is not new and other researchers have investigated it previously. However, incremental slicing based on data-dependence types is novel. Slices can be computed incrementally based on parameters other than data-dependence types; for example, slices can be computed one procedure at a time or one level of dependence at a time as can be done in the CANTO environment [Antoniol et al. 1997].

Horgan 1990]. 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 an execution that fails. Thus, for debugging, 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[11] 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 considered the data dependences in the order shown in Table IX. If no or very few data dependences occurred for a type, we did not compute a separate increment for that type. We added data-dependence types for an increment as long as the cumulative additional data dependences for the increment was less than 1% of the data dependences in the program. When the cumulative additional data dependence for the increment exceeded 1%, we went to the next increment. We use the names t1, t2, ... , t54 to refer succinctly to the 54 types of data dependences; Table IX 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 III. For each subject, we determined, based on the distribution of data-dependence types, the appropriate incremental slices to compute. We considered the types in the order shown in Table IX. To select the increments, we used the same procedure described above. However, for ease of presenting the data, we limited the number of increments. Therefore, instead of using 1% (of data dependences) as a threshold, we used 5%, and obtained fewer increments. Table X 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 X. 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 from each procedure in the program and computed a slice for each of those criteria. Thus, the number of slices

[11]The faults are naturally occurring—they occurred during the course of development and maintenance of the subject.

Table IX. 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

Table X. 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}

computed for a subject was five times the number of procedures in the subject. For example, for `bison`, we computed 655 slices ($131 * 5$). We then examined the differences in the sizes of the increments.

Results and analysis. Table XI 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 increment contained 126 statements in addition to the 38 from the previous increment.

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

Table XI. Sizes of incremental approximate dynamic slices for 15 versio ns of `space` for fault detection.

	Inc. 1		Inc. 2		Inc. 3		Inc. 4		Inc. 5	
Version	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**
2	181	181	0	181	163	163	7	351	**222**	**573**
3	172	172	0	0	161	161	7	340	130	470
4	179	179	0	179	159	338	7	345	137	482
5	195	195	0	195	162	357	7	364	164	164
6	187	187	0	187	163	350	7	357	**170**	**527**
7	205	205	0	205	187	392	7	399	204	603
8	187	187	0	187	163	350	7	357	**164**	**521**
9	188	188	0	188	157	345	7	352	182	534
10	**189**	**189**	0	189	161	350	7	357	170	527
11	56	56	0	56	128	184	2	186	**76**	**262**
12	173	173	0	173	**152**	**325**	6	331	131	462
13	94	94	0	94	**127**	**221**	4	225	95	320
14	38	38	0	38	126	164	2	166	**61**	**227**
15	**185**	**185**	0	185	150	335	7	342	149	491

	Inc. 6		Inc. 7		Inc. 8		Inc. 9	
Version	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size	Inc size	Cum size
1	50	280	12	292	12	304	1	305
2	73	646	82	728	22	750	1	751
3	**89**	**559**	85	644	26	26	1	671
4	**95**	**577**	78	655	26	681	1	682
5	**127**	**655**	114	769	28	797	1	798
6	128	655	74	729	29	758	1	759
7	**81**	**684**	98	782	24	806	1	807
8	123	644	78	722	26	748	1	749
9	**86**	**620**	110	730	23	753	1	754
10	64	591	93	684	27	711	1	712
11	96	358	47	405	40	445	1	446
12	62	524	82	606	24	630	1	631
13	58	378	82	460	19	479	1	480
14	32	259	12	271	12	283	1	284
15	87	578	98	676	28	704	1	705

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 because it lets the testers examine, at a time, a set of potentially fault-containing statements that is smaller than a complete slice. Intuitively, it is simpler

[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.

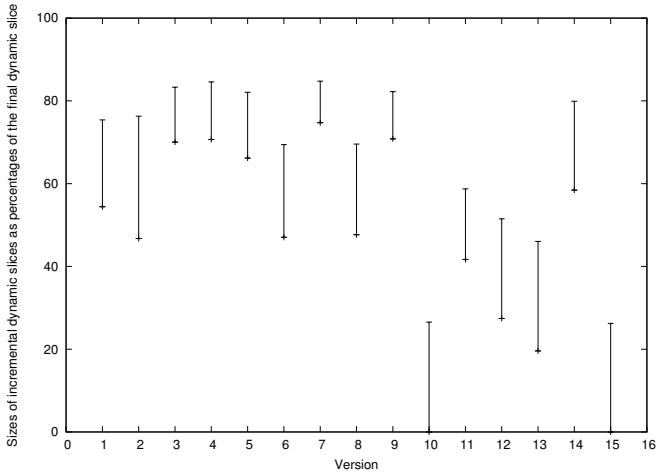

Fig. 10. Size of the dynamic slice increment that contained the fault, for each of the 15 versions of `space`.

to examine N increments of X statements each (as long as the statements are related and not randomly selected) than to examine a single set of statements whose size is $N * X$. Moreover, if the fault occurs in an increment computed before the last increment, testers can avoid examining the statements that would appear only in the successive increments. Finally, testers can use their knowledge of the failure to compute the slice increments, which would further increase the chances that the fault would appear in an increment computed before the final increment.

The data in Table XI 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.

Figure 10 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 11 presents the results of the second study, in which we computed the slices listed in Table X. The vertical axis represents the sizes of the slices as percentages of the number of statements in the program. Each segmented bar in Figure 11 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

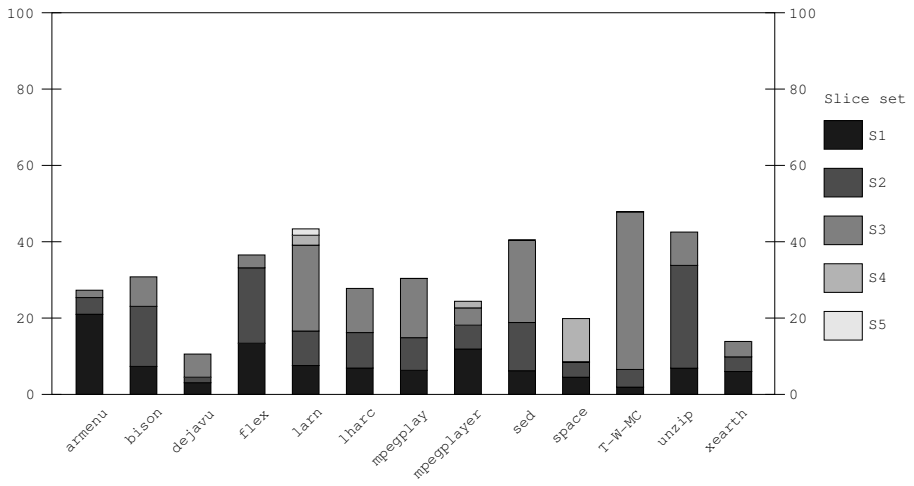

Fig. 11. Increase in the sizes of the slices for the slice increments listed in Table X. For each subject, the segmented bar illustrates the average increase in the slice size from one increment to the next.

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, computed for data-dependence types t1 through t3, are, on average, 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 11. 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 sizes 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 potentially augments

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 of our 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 [1991] extend the traditional data-flow testing techniques [Frankl and Weyuker 1988; Rapps and Weyuker 1985] 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.

Table XII. Comparison of our classification with those of Ostrand and Weyuker [1991] and Merlo and Antoniol [1999; 2000].

	Ostrand and Weyuker's Classification	Merlo and Antoniol's Classification	Our Classification
Definition/Use type	definite, possible	definite, possible	direct, single alias, multiple alias
Rd type	[G, GR], [GY, GYR], [Y, YR] [G, GY, GR, GYR, Y, YR]	[G, GY, GR, GYR] [G, GY, GR, GYR, Y, YR]	G, GY, GR, GYR, Y, YR
Number of data-dependence types	4	2	54

Merlo and Antoniol [1999; 2000] 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. Merlo and Antoniol do not mention whether they consider a single-alias access to be a definite access.

Table XII summarizes how our classification compares with Ostrand and Weyuker's and Merlo and Antoniol's. In considering rd types for data dependences, both Ostrand and Weyker and Merlo and Antoniol group several rd types from our classification; such types are shown in square brackets in Table XII. For example, for one of the data-dependence types in Ostrand and Weyuker's classification, the rd type is G or GR; for another type, it is GY or GYR.

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.

Other research is related to some aspects of our work. However, none of that research has the same goal as ours—classifying data dependences and evaluating

the usefulness of the classification. We discuss some of that research in the rest of this section.

Some researchers have discussed how slices can be computed incrementally. For example, CANTO [Antoniol et al. 1997], a program understanding and architecture recovery tool, lets the user compute a slice one step at a time, in an incremental manner. At each step, the tool augments the slice by considering one step of data dependences, control dependences, or function calls. The user can inspect the newly added statements before computing the next increment. Our approach provides an alternative way of computing incremental slices—based on data-dependence types.

Marré and Bertolino [2003] 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. Similar to our approach for data-flow testing, Marré and Bertolino use properties of data dependences to reduce the number of testing requirements that are considered for coverage. However, the properties that we consider (types and spans of data dependences) are different from the properties that they consider (implications among data dependences).

Other researchers (e.g. [Canfora et al. 1998; Harman and Danicic 1997]) 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 achieving the reduction.

Finally, several researchers have considered the effects of pointers on program slicing and have presented approaches for performing slicing more effectively in the presence of pointers (e.g. [Agrawal et al. 1991; Atkinson and Griswold 1998; Binkley 1993; Binkley and Lyle 1998; Liang and Harrold 1999b]). Some researchers have also evaluated the effects of the precision of the pointer analysis on subsequent analyses, such as the computation of data dependences (e.g., [Tonella 1999; Tonella et al. 1999]) and program slicing (e.g., [Bent et al. 2000; Liang and Harrold 1999a; Shapiro and Horwitz 1997]). 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.

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 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 the second application, we presented a new slicing approach in which slices are computed by traversing data dependences selectively, based on their types. The new slicing approach 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 presented the results of three empirical studies that we performed using a set of C subjects. The overall goal of these studies was to evaluate the usefulness of the classification scheme. In the first study, we investigated the occurrences of data-dependence types in practice and whether such occurrences can be used to characterize programs. The results of the study indicated that, although a number of different data-dependence types can occur, the five most-frequently-occurring data-dependence types can account for as many as 90% of the data dependences in the programs (Figure 5). Data about the most-frequently-occurring types in a program (Table V) 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.

Because our first study indicated that many data-dependence types can occur in practice, we conducted two more studies to evaluate the two applications of the classification. In the second empirical study, we showed how our approach can be applied to data-flow testing. The results of the study suggest that the approach can be 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 third empirical study, we investigated how incremental approximate dynamic slices can be used to reduce fault-detection effort. Using incremental slicing, testers can examine a much smaller fault space at a time, which can speed up the process of locating a fault. Moreover, our results indicated that testers need not examine the entire fault space (Table XI, Figure 10). We further evaluated 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. The outcome of the study shows that incremental slicing can be effective in reducing the slice size across the subjects considered (Figure 11), 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 potential area of future work is 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. Future research could also investigate whether any relations exist between occurrences of data-dependence types and presence of errors in industrial systems. 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.

REFERENCES

Agrawal, H., DeMillo, R. A., and Spafford, E. H. 1991. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV 91), Victoria, British Columbia, Canada*. ACM Press, New York, NY, USA, 60–73.

Agrawal, H. and Horgan, J. R. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 90), White Plains, New York*. ACM Press, New York, NY, USA, 246–256.

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

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Tech. Rep. 94-19, University of Copenhagen. May.

Antoniol, G., Fiutem, R., Lutteri, G., Tonella, P., Zanfei, S., and Merlo, E. 1997. Program understanding and maintenance with the CANTO environment. In *Proceedings of the International Conference on Software Maintenance (ICSM 97), Bari, Italy*. IEEE Computer Society, Los Alamitos, CA, USA, 72–83.

Aristotle Research Group. 2000. Aristotle: Software engineering tools. `http://www.cc.gatech.edu/aristotle/`.

Atkinson, D. C. and Griswold, W. G. 1998. Effective whole-program analysis in the presence of pointers. In *Proceedings of ACM SIGSOFT 6^{th} International Symposium on the Foundations of Software Engineering (FSE 98), Lake Buena Vista, Florida*. IEEE Computer Society, Los Alamitos, CA, USA, 46–55.

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

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

Binkley, D. W. 1993. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum, Orlando, Florida*. 261–268.

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

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

Clarke, L. A., Podgurski, A., Richardson, D. J., and Zeil, S. J. 1989. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng. 15,* 11 (Nov.), 1318–1332.

Frankl, P. G. and Weiss, S. N. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng. 19,* 8 (Aug.), 774–787.

Frankl, P. G. and Weyuker, E. J. 1988. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. 14,* 10 (Oct.), 1483–1498.

Frankl, P. G. and Weyuker, E. J. 1993. Provable improvements on branch testing. *IEEE Trans. Softw. Eng. 19,* 10 (Oct.), 962–975.

Harman, M. and Danicic, S. 1997. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension (IWPC 97), Dearborn, Michigan*. IEEE Computer Society, Los Alamitos, CA, USA, 70–79.

Harrold, M. J. and Ci, N. 1998. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 98), Kyoto, Japan*. IEEE Computer Society, Los Alamitos, CA, USA, 74–83.

Harrold, M. J. and Rothermel, G. 1997. Aristotle: A system for research on and development of program-analysis-based tools. Tech. Rep. OSU-CISRC-3/97-TR17, The Ohio State University. Mar.

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

Harrold, M. J. and Soffa, M. L. 1994. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst. 16,* 2 (Mar.), 175–204.

Horwitz, S. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst. 19,* 1 (Jan.), 1–6.

Horwitz, S., Reps, T., and Binkley, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12,* 1 (Jan.), 26–60.

Korel, B. and Laski, J. 1988. Dynamic program slicing. *Inf. Process. Lett. 29,* 3 (Oct.), 155–163.

Landi, W. and Ryder, B. G. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 92), San Francisco, California*. ACM SIGPLAN Notices, vol. 27. ACM Press, New York, NY, USA, 235–248.

Laski, J. W. and Korel, B. 1983. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. 9,* 3 (May), 347–354.

Liang, D. and Harrold, M. J. 1999a. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European engineering conference held jointly with the th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 99), Toulouse, France*. LNCS, vol. 1687. Springer-Verlag, London, UK, 199–215.

Liang, D. and Harrold, M. J. 1999b. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 99), Oxford, England*. IEEE Computer Society, Los Alamitos, CA, USA, 421–432.

Liang, D. and Harrold, M. J. 2001. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of the 8th Static Analysis Symposium (SAS 01), Paris, France*. LNCS, vol. 2126. Springer-Verlag, London, UK, 279–298.

Marré, M. and Bertolino, A. 2003. Using spanning sets for coverage testing. *IEEE Trans. Softw. Eng. 29,* 11 (Nov.), 974–984.

Merlo, E. and Antoniol, G. 1999. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *Proceedings of CASCON 99, Mississauga, Ontario, Canada*. 173–186.

Merlo, E. and Antoniol, G. 2000. Pointer sensitive def-use pre-dominance, post-dominance and synchronous dominance relations for unconstrained def-use intraprocedural computation. Tech. Rep. EPM/RT-00/01, Ecole Polytechnique of Montreal. Mar.

Ntafos, S. 1984. On required elements testing. *IEEE Trans. Softw. Eng. 10,* 6 (Nov.), 795–803.

Ntafos, S. 1988. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng. 14,* 6 (June), 868–874.

Orso, A., Liang, D., Sinha, S., and Harrold, M. J. 2002. A framework for understanding data dependences. Tech. Rep. GIT-CC-02-13, College of Computing, Georgia Institute of Technology. Mar.

Orso, A., Sinha, S., and Harrold, M. J. 2003. Understanding data dependences in the presence of pointers. Tech. Rep. GIT-CERCS-03-10, College of Computing, Georgia Institute of Technology. May.

Ostrand, T. J. and Weyuker, E. J. 1991. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the symposium on Testing, analysis, and verification (TAV 91), Victoria, British Columbia, Canada*. ACM Press, New York, NY, USA, 74–86.

Pande, H., Landi, W., and Ryder, B. G. 1994. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. Softw. Eng. 20,* 5 (May), 385–403.

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

Rapps, S. and Weyuker, E. J. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. 11,* 4 (Apr.), 367–375.

Reps, T., Horwitz, S., Sagiv, M., and Rosay, G. 1994. Speeding up slicing. In *Proceedings of the 2^{nd} ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 94), New Orleans, Louisiana*. ACM Press, New York, NY, USA, 11–20.

Shapiro, M. and Horwitz, S. 1997. The effects of the precision of pointer analysis. In *4^{th} International Static Analysis Symposium (SAS 97), Paris, France*. LNCS, vol. 1302. Springer-Verlag, London, UK, 16–34.

Sinha, S., Harrold, M. J., and Rothermel, G. 1999. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21^{st} International Conference on Software Engineering (ICSE 99), Los Angeles, California*. IEEE Computer Society Press, Los Alamitos, CA, USA, 432–441.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23^{rd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 96), St. Petersburg Beach, Florida*. ACM Press, New York, NY, USA, 32–41.

Tonella, P. 1999. Effects of different flow insensitive points-to analyses on DEF/USE sets. In *Proceedings of the 3^{rd} European Conference on Software Maintenance and Reengineering (CSMR 99), Amsterdam, The Netherlands*. IEEE Computer Society Press, Los Alamitos, CA, USA, 62–69.

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

Weiser, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10,* 4 (July), 352–357.

Received Month Year; revised Month Year; accepted Month Year