Technical Report GIT-CC-02-13, March 2002

A Framework for Understanding Data Dependences*

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

Abstract

Identifying and understanding data dependences is important for a variety of software-engineering tasks. The presence of pointers, arrays, and dynamic memory allocation introduces subtle and complex data dependences that may be difficult to understand. In this paper, we present a refinement of our previously developed classification that also distinguishes the types of memory locations, considers interprocedural data dependences, and further distinguishes such data dependences based on the kinds of interprocedural paths on which they occur. This new classification enables reasoning about the complexity of data dependences in programs using features such as pointers, arrays, and dynamic memory allocation. We present an algorithm for computing interprocedural data dependences according to our classification. To evaluate the classification, we compute the distribution of data dependences for a set of real C programs and we discuss how the distribution can be useful in understanding the characteristics of a program. We also evaluate how alias information provided by different algorithms, varying in precision, affects the distribution. Finally, we investigate how the classification can be exploited to estimate complexity of the data dependences in a program.

keywords: Data dependences, data flow, static analysis, pointers

1 Introduction

Many software-development tasks rely on the use and understanding of the data dependences in a program. Such dependences can be identified by computing definition-use associations. A *definition-use association* (DUA) relates a statement that assigns a value to a memory location (*definition*) to a statement that references that memory location without modifying it (*use*). For example, to reverse engineer a program, a maintainer must know and understand the program's DUAs. For another example, to locate a fault during debugging, a developer often exploits knowledge about the program's DUAs, either manually or through a tool [1, 18]. For a third example, to test a program, a tester may want to ensure that certain DUAs are executed under test [5]. More generally, the results of activities such as verification, testing, maintenance, debugging, understanding, and reverse engineering depend significantly on understanding a program's DUAs.

DUAs can be computed using data-flow analysis algorithms [2]. However, in the presence of language constructs such as pointers, arrays, and dynamic-memory allocation, DUAs computed by existing data-flow analyses may be difficult for use in software-engineering tasks. One difficulty is that the data-flow analysis can compute a large number of DUAs. Analyzing these DUAs in an arbitrary order may not be an effective way to accomplish the goals of a particular task. For example, consider a tester who must meet a

*This work was supported in part by grants to Georgia Tech from Boeing Commercial Airplanes and NSF under awards CCR-9707792, CCR-9988294, and CCR-0096321.

data-flow coverage goal, with limited resources, and has no way of estimating the cost of covering different DUAs. A second difficulty is that the data-flow analysis can compute spurious DUAs. In general, this spurious information will negatively affect the task and, in some cases, may even prevent the task from being performed successfully. For example, spurious information about dependences among modules may lead to a poor understanding of a system during reverse engineering or maintenance. A third difficulty is that, even if a DUA is not spurious, in the presence of pointers, arrays, dynamic-memory allocation, branching, and loops, the conditions under which the DUA actually occurs during execution may be too complex for the DUA to be useful for the task. For example, the conditions for the DUA may be too difficult for a debugger to effectively use the dependence information while trying to localize a fault.

The overall goal of this research is to improve software-engineering tasks through a better understanding of data dependences. To this end, we have investigated different properties of a DUA. These properties form a framework that can be used to characterize and understand DUAs, and overcome the difficulties in using DUAs to support software-engineering tasks. For example, characterizing DUAs within a program according to their estimated complexity may provide useful information for identifying parts of the program that can be more difficult to maintain.

Ostrand and Weyuker present a classification of DUAs [17] that accounts for the effects of pointers on definitions and uses. Based on the classification, they define new test-adequacy criteria. In their work, however, Ostrand and Weyuker did not consider the effects of arrays and dynamic-memory allocation on the type of a DUA, and they did not classify DUAs that cross procedure boundaries (i.e., *interprocedural*). Moreover, they presented no empirical data to evaluate their classification. Merlo and Antoniol [14] present techniques to identify implications between nodes and data dependences, and distinguish definite and possible definitions and uses. They define relations to estimate the data dependences whose coverage is implied by the coverage of a node. However, they do not classify DUAs or investigate how classifying DUAs can be used to support software-engineering tasks.

In previous work, we presented a preliminary classification of DUAs that is finer grained than Ostrand and Weyuker's [15], and we investigated the application of our classification to program slicing [16].

In this paper, we present a refinement of our initial classification. In refining the classification, we investigate further the properties of the three elements that comprise a DUA: the definition, the use, and the path(s) between the definition and the use. There are two main features of the refined classification. First, it distinguishes the types of memory locations that are defined or used—scalar variables, arrays, and heap-allocated memory. Second, it applies the classification to interprocedural DUAs, and considers an additional property of such DUAs—the types of the interprocedural paths on which the DUAs occur. The result is a classification of DUAs along four dimensions. The first two dimensions characterize definitions and uses by distinguishing (1) types of accesses to a memory location and (2) types of memory locations. The last two dimensions characterize the paths between definitions and uses by distinguishing occurrences of (1) redefinitions along the paths and (2) sequences of procedure calls and returns along the paths. We present an efficient algorithm for computing interprocedural DUAs and classifying them along the four dimensions.

The four dimensions form a framework that can be used to classify, compare, prioritize, and understand DUAs. By enumerating the properties of a DUA in a systematic way, the framework expresses clearly various factors that affect a DUA; understanding these factors is essential to understanding a DUA. We investigate two applications of the framework and present empirical results for one of them.

First, we apply the framework to classify each DUA in a program and obtain a distribution of the types of DUAs in the program. We present empirical results that illustrate the distribution of DUAs for a set of C programs and discuss how the distribution can be useful in understanding the characteristics of a program. We also investigate the effects of different alias-analysis algorithms on the distribution of data-dependence types.

Second, we apply the framework to estimate the complexity of a DUA. Such an estimate can be useful for assessing the effort required to understand, or develop test cases to cover, a DUA. The estimated complexity of a DUA can also be used to measure the data-flow complexity of the statements and procedures in which the associations occur. We discuss a parameterized approach to estimate the data-flow complexity of the code at different levels. The algorithm is implemented in a visualization tool that provides an overview of the spectrum of the data-flow complexity in the program and can be tuned for investigating or supporting different software-engineering tasks.

2 Classifying DUAs

In the presence of constructs such as pointers, arrays, and dynamic-memory allocation, it is possible to distinguish different types of DUAs. Based on the characteristics of definitions and uses, and on the types of paths between definitions and uses, we can classify DUAs along several dimensions.

2.1 Definitions and Uses

We characterize definitions and uses along two dimensions: (1) the type of access of a memory location and (2) the type of memory location that is accessed. The first dimension accounts for the presence of pointer dereferences, whereas the second dimension accounts for the effects of different types of memory locations. In either case, definitions and uses can be either definite or possible. A *definite* definition or use is one that occurs each time that the statement containing the definition or use executes; a *possible* definition or use may occur in some execution of the statement and not in others.

2.1.1 Access type

In programs with pointers, memory locations can be accessed not only through variable names, but also through pointer dereferences. For example, in the program in Figure 1, either `x` or `y` may be accessed indirectly through `*p` at statement 12.[1] Alias information provided by alias-analysis algorithms can be used to determine the memory locations that may be accessed through a pointer dereference. However, because precisely determining such memory locations is an undecidable problem, an alias-analysis algorithm can provide only a conservative estimate of the alias relations.[2] Thus, when memory locations are accessed through pointer dereferences, a data-flow analysis can compute spurious DUAs. In addition, accessing memory locations through pointer dereferences can complicate the conditions under which a DUA occurs. For example, at statement 12 in Figure 1, `x` is defined only if the condition in statement 5 evaluates true.

We identify three types of accesses of memory locations that can occur in the presence of pointers: direct, single-alias, and multiple-alias. A *direct* access involves no pointer dereference. A *single-alias* access

[1] The phenomenon that a memory location may be accessed using more than one name is often referred to as *aliasing*.

[2] Existing alias-analysis algorithms (e.g., [3, 9, 10, 20]) vary in the efficiency and the precision with which they compute the alias relations and in the kinds of alias information they compute.

```
1. int g;                    10. f1(int *p) {
2. main() {                  11.    if(*p<0) {
3.    int x,y,*t;            12.      *p=g;
4.    read(&x,&y,&g);        13.    }
5.    t=(g>0)?&x:&y;         14. }
6.    f1(t);                 15. f2(int *q) {
7.    f2(&x);                16.    print(*q);
8.    print(y);              17. }
9. }
```

Figure 1: An example program to illustrate data dependences in the presence of aliasing.

is performed through a dereference of a pointer that can point to a single memory location. A *multiple-alias* access is performed through a dereference of a pointer that can point to multiple memory locations.[3] For example, in Figure 1, the definition in statement 5 involves a direct access, whereas the definition in statement 12 involves a multiple-alias access; the use in statement 16 involves a single-alias access.

2.1.2 Memory-location type

The presence of arrays and heap-allocated memory locations can further complicate memory accesses. Memory locations within an array are accessed through an index, whose value may depend on the computation results of other statements. Thus, it may be impossible to determine statically the exact value for each index. Due to this difficulty, many program-analysis techniques treat an array as a single memory location. This approach causes data-flow analysis to be imprecise and generate spurious results because definitions and uses of different elements of the same array cannot be distinguished.

A similar situation occurs for heap locations. The memory block returned by an allocation function can be used in the same way as an array. Thus, many program-analysis techniques cannot distinguish the different locations in such a block. Even worse, a heap-allocating statement can be executed many times, each of which can return a different memory block. However, because it may not be possible to determine statically the number of times such a statement can execute, most program analyses treat all the memory blocks returned by the statement as one memory location. Thus, these program analyses can compute very imprecise information for heap locations.

To account for the differences in DUAs caused by various types of memory locations, we distinguish definitions and uses based on these types. We distinguish three types of memory locations: scalars, arrays, and heap locations.

If the alias information is provided by parameterized pointer analysis [11], the types of memory locations may be unclear because a memory location may be identified using different names in different procedures. A parameterized pointer analysis uses symbolic names to identify memory locations whose addresses are passed into a procedure through formal parameters. For example, in function `f1()` in Figure 1, parameterized pointer analysis may use symbolic name "?1" to identify `x` or `y`, whose addresses are passed into `f1()` through formal parameter `p`. In different calling contexts, a symbolic name may be used to identify different memory locations, each of which may have a different type. Our algorithm classifies the types for those memory locations that are accessed through symbolic names by mapping the symbolic names, when possible, to the actual memory locations that they represent. If the algorithm fails to map a symbolic name to an actual

[3]Note that a single-alias access to a memory location is considered a possible definition (or use) of that memory location. This occurs because of the limitations of static analysis in approximating the dereferenced memory locations.

memory location, it simply classifies the name as a symbolic name.

2.2 Paths Between Definitions and Uses

A *path* in a control-flow graph (CFG)[4] is a sequence of nodes $(n_1, n_2, \ldots, n_k)$, $k \geq 1$, such that, if $k \geq 2$, for $i = 1, 2, \ldots, k-1$, (n_i, n_{i+1}) is an edge in the CFG. In a program with procedures, a path can contain procedure calls and returns.

The paths between a definition d and a use u determine whether a DUA involving d and u can occur. In the presence of arrays, heap locations, and pointer dereferences, it may be unclear whether d can reach u along a particular path and under what condition d can reach u. The situation can be further complicated if the path traverses multiple procedures.

To account for the effects of paths on DUAs, we distinguish paths between definitions and uses based on two properties. The first property considers occurrences of redefinitions of a variable along the paths—we call this property the reaching-definition type. The second property considers the occurrences of sequences of calls and returns along the paths—we call this property the path type.

2.2.1 Reaching-definition type

We classify each path from a definition to a use into one of three types with respect to a memory location v. A path $(d, n_1, \ldots, n_k, u)$ is *green* with respect to v if the n_i contain no definition of v; a path is *yellow* with respect to v if the n_i contain at least one possible definition, but no definite definition, of v; a path is *red* with respect to v if the n_i contain at least one definite definition of v.

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

Typically, there is a set of paths from a definition of memory location v to the use of v. Thus, we classify a DUA of v based on the colors of this set of paths with respect to v. The three colors yield six combinations: all green, all yellow, green and yellow, green and red, yellow and red, and green and yellow and red.[5] We refer to these types as the *reaching-definition types* (RD types) of a DUA.

2.2.2 Path type

We also classify a path that crosses procedure boundaries according to the sequence of call and return edges that appear in the path [13]. Such paths are represented in an interprocedural control-flow graph (ICFG).[6] Figure 2 shows the ICFG for the example. A path is a *same-level path* (SLP) if each call edge in the path is matched with a return edge. An SLP represents an execution sequence that begins and ends in the same procedure. A path is an *unbalanced-left path* (ULP) if it contains at least one call edge that is not

[4] A *control-flow graph* for procedure P contains nodes, which represent statements in P, and edges, which represent potential flow of control among the statements. Each call site in P is represented using a call node and return node.

[5] The combination of all red is not considered because it corresponds to the case in which no path between the definition and the use propagates the definition; therefore, the definition and the use do not form a DUA.

[6] An *interprocedural control-flow graph* for a program consists of the CFG for each procedure in the program. Each call node is connected to the entry node of the CFG for the called procedure by a call edge; the exit node of CFG for the called procedure is connected to the corresponding return node by a return edge.

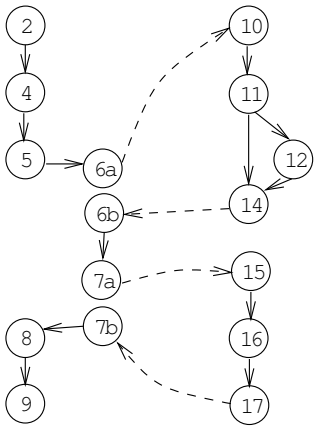

Figure 2: The ICFG for the program of Figure 1.

matched with a return edge. A ULP represents an execution sequence in which some procedure calls have not completed. A path is an *unbalanced-right path* (URP) if it contains at least one return edge that is not preceded by a matching call edge. A URP represents an execution sequence in which some procedure calls complete such that the sequence that led to the invocations of those procedures is not part of the path. Finally, a path is an *unbalanced-right-left path* (URLP) if it contains an unbalanced-right subpath followed by an unbalanced-left subpath. For example, in the Figure 2, (4, 5, 6a, 10, 11, 12) is an unbalaned-left path and (12, 14, 6b, 7a, 15, 16) is an unbalanced-right-left path.

For a same-level path p that connects a definition of memory location v to a use of v, we consider a special case: none of the calls in p invoke procedures that can redefine v. In such a case, to determine if the definition may reach the use through p, we need to consider only the nodes in the procedure that contains the definition and the use. We classify p as an *intraprocedural path* (IPP) with respect to v. Hereafter, we use SLP to refer to a same-level path that is not an IPP with respect to v.

As mentioned in Section 2.1.1, there may be a set of paths from a definition of memory location v to a use of v. Thus, we classify a DUA of v based on the set of path types for the paths that connect the definition to the use. We refer to these types as the *path types*.

3 Framework for Understanding DUAs

Figure 3 summarizes the four dimensions along which we classify DUAs: access type, memory-location type, RD type, and path type. The set of dimensions that we consider is not intended to be exhaustive. For example, the number of paths between a definition and a use can be an additional dimension to consider (although very expensive to compute in practice). The goal of the classification that we define is to provide enough information about DUAs and still be practical. Therefore, based on our past experience [15, 16], we defined the classification presented in this paper, which allows for a fine-grained analysis of data dependences and can be efficiently implemented in a tool.

Because different software-engineering tasks may use data dependences in different ways, different kinds of information about dependences may be relevant based on the task. Separating the properties of DUAs along four dimensions in a systematic way lets us focus on the different aspects of an association separately,

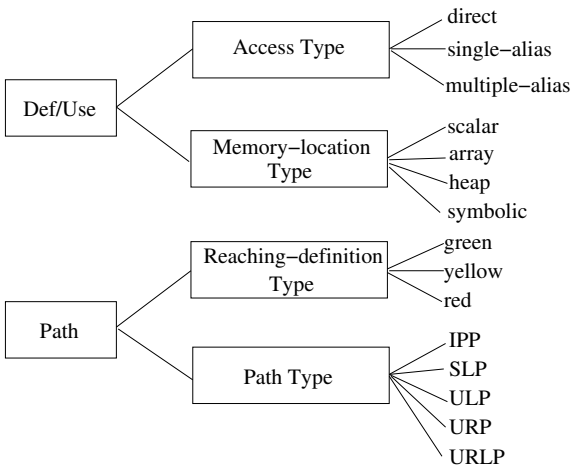

Figure 3: Dimensions of the DUA classification.

and therefore, provides us with a framework for studying and understanding DUAs in different contexts and considering different needs. More precisely, we use the framework to investigate data dependences and understand the data-flow characteristics of the programs in which the data dependences occur.

One application of the framework is to classify each DUA in a program along the four dimensions and obtain a distribution of the types of associations in the program. The occurrences—in predominant or negligible numbers—or non-occurrences of various types of DUAs can provide insight into the data-flow characteristics of programs. Such distributions can also help identify patterns that are useful for program comprehension of parts of the program where specific types of DUAs occur.

Another application of the framework is to compute an estimate of the complexity of a DUA, and based on that, an estimate of the data-flow complexity of a program or parts of a program. The estimated complexity of a DUA can indicate the effort required to develop test cases to cover that association. It can also indicate if a DUA is likely to be infeasible. To estimate the complexity of a DUA, first, a score is assigned to each value that occurs along a dimension; this score reflects the intuitive complexity of the DUA along that dimension. Then, the scores for each dimension can be combined to compute a complexity value for a DUA.

Consider, for example, the access type of a DUA. A direct access is the simplest type of access because, to understand a direct access, we need to examine only the statement where the access occurs. Moreover, such an access occurs each time that the statement containing the access executes. A multiple-alias access, on the other hand, is much more difficult to understand because we must also understand the alias relations that hold at the statement where the access occurs, which requires examining other parts of the programs—parts where the alias relations are established. Moreover, a multiple-alias access may or may not occur on any given execution of the statement containing the access depending on the alias relations that hold in that execution. A single-alias access falls in between these two in terms of complexity because, although it requires understanding the alias relations, only a single alias exists at the access point.

Similarly, we can assign complexity scores to the values along other dimensions.

Based on the estimated complexity of a DUA, we can estimate the data-flow complexity of a program statement. The underlying intuition is that, to understand a statement (from a data-flow standpoint), we have to understand all the DUAs that involve the statement. In other words, all the DUAs such that the

definition and/or the use occur at the statement contribute to defining the data-flow complexity of the statement. Therefore, for each statement, we can combine the estimated complexity values of such DUAs and compute an overall complexity estimate for the statement. This estimate can be further aggregated in different ways and used for different purposes.

By varying the scores assigned to the values along different dimensions, the framework lets us tune the complexity value according to the context. For example, if we assign a relatively high score to accesses of heap locations and compute the complexity estimate for each statement in the program, the resulting complexity spectrum would highlight statements in the program that use dynamic memory. We can further tailor the complexity computation by assigning weights to the values. The weight reflects the contribution of a particular value to the overall complexity of the DUA. For example, we can assign higher weights to RD types and thus have the values along that dimension contribute more to the complexity of a DUA than other dimensions.

Similarly, we can compute the data-flow complexity of a statement in a flexible way. To compute the data-flow complexity of statement s, we can consider (1) DUAs such that the use occurs in s and the definition elsewhere, (2) DUAs such that the definition occurs in s and the use elsewhere, or (3) both. For example, when performing data-flow testing to satisfy the *all-defs* criterion [5], we can highlight statements that contain complex definitions by using option (2).

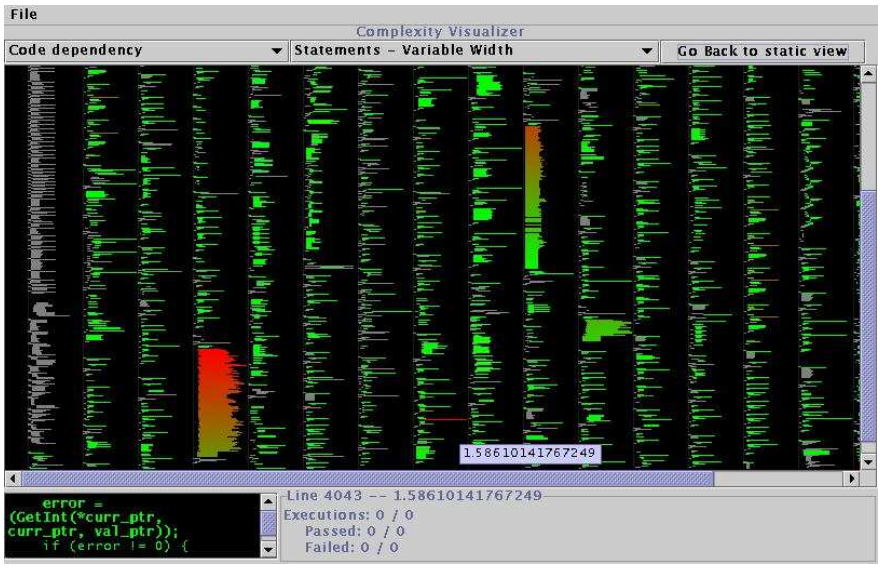

Figure 4: Screenshot of the complexity-visualization tool.

We have implemented the algorithm for computing the data-flow complexity of each statement and a tool to visualize the complexity spectrum of the statements in a program. The tool uses a SeeSoft [4] view of the program, and shows statements using different colors based on the estimated complexity of the statements. The tool, shown in Figure 4, provides a convenient way of modifying the values of the parameters of the algorithm dynamically, through its GUI, and shows the complexity information at different levels of abstraction: DUA, statement, and procedure. We are currently investigating the usefulness of the tool in supporting software-engineering tasks.

4 Computing Interprocedural DUAs

In this section, we present an algorithm that computes and classifies DUAs. Our algorithm extends Harrold and Soffa's algorithm [7], which computes interprocedural DUAs in two phases. In the intraprocedural phase, the algorithm analyzes each procedure in the program and computes information that is local to the procedure. The local information consists of DUAs that do not cross procedure boundaries, and information that is needed for the interprocedural phase. In the interprocedural phase, the algorithm constructs an interprocedural flow graph and computes DUAs that cross procedure boundaries.

Before computing the DUAs, the algorithm computes two sets of memory locations for each procedure P: (1) GMOD(P), which contains nonlocal memory locations that are modified by P or a procedure called directly or indirectly from P, and (2) GREF(P), which contains nonlocal memory locations that are referenced by P or a procedure called directly or indirectly from P. If a nonlocal memory location is identified using a symbolic name within P, the GMOD and GREF sets for P contain the symbolic name instead of the memory location. For example, considering the example in Figure 1 and assuming that "?1" is used to identify `x` or `y` in `f1()`, the GMOD set for `f1()` contains "?1", and the GREF set contains "?1" and `g`. In the following, we assume that our algorithm uses parameterized pointer information.

4.1 Intraprocedural Phase

In previous work [15], we presented an algorithm that uses a set of data-flow equations to compute and classify DUAs for individual procedures. We extend the algorithm to also compute information that is needed to build the IFG in the interprocedural phase.

For each node n in the CFG of a procedure P, the algorithm computes six sets—three IN sets for the entry of n and three OUT sets for the exit of n. $IN_d(n)$ (or $OUT_d(n)$) contains the set of definitions that can reach the entry (or the exit) of n through a green path. $IN_p(n)$ (or $OUT_p(n)$) contains the set of definitions that can reach the entry (or the exit) of n through a yellow path. $IN_k(n)$ (or $OUT_k(n)$) contains the set of definitions that can reach the entry (or the exit) through a red path.

The algorithm computes each $IN(n)$ set by combining the OUT sets of each predecessor of n, and computes each $OUT(n)$ using $IN(n)$ and the information associated with n. Let $GEN(n)$ be the set of definitions that are created at n, $KILL_d(n)$ be the set of definitions that are definitely killed at n, and $KILL_p(n)$ be the set of definitions that are possibly killed at n. The algorithm computes the $OUT(n)$ sets using the following data-flow equations:

$$OUT_d(n) = GEN(n) \cup (IN_d(n) - KILL_d(n) - KILL_p(n)) \quad (1)$$

$$OUT_p(n) = (IN_d(n) \cap KILL_p(n)) \cup (IN_p(n) - KILL_d(n)) \quad (2)$$

$$OUT_k(n) = ((IN_d(n) \cup IN_p(n)) \cap KILL_d(n)) \cup IN_k(n) \quad (3)$$

The algorithm solves these equations using a worklist. For each definition d that reaches a node n, the algorithm determines the RD types for d as follows: if $d \in IN_d(n)$, the RD types of d includes "green"; if $d \in IN_p(n)$, the RD types of d includes "yellow"; if $d \in IN_k(n)$, the RD types of d includes "red".

The extended algorithm creates placeholder definitions and uses before computing the intraprocedural DUAs. At the entry node of each procedure P, the algorithm creates a placeholder definition for each

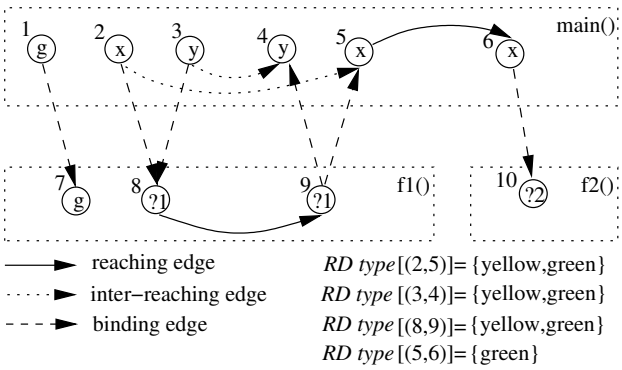

Figure 5: The IFG for the program of Figure 1.

memory location that appears in either GREF(P) or GMOD(P). At the exit node of each procedure P, the algorithm creates a placeholder use for each memory location that appears in GMOD(P). At the call node for each call site c that calls procedure P, the algorithm creates a placeholder use for each memory location that appears in GREF(P) or GMOD(P); at the corresponding return node, the algorithm creates a placeholder definition for each memory location that appears in GMOD(P).[7] For example, the GREF set for `f1()` contains "?1" and `g`. Therefore, the algorithm creates definitions of "?1" and `g` at the entry node of the CFG for `f1()`, and uses of `x`, `y`, and `g` at the call node for statement 6 in the CFG for `main()`.

The extended algorithm also processes a call site c in a different way than our previous algorithm. For each definition d of v that can reach the entry of c, if v appears in the GMOD set of the procedure invoked at c, the types of paths through which d reaches the corresponding return site cannot be determined without analyzing the called procedure. Therefore, the algorithm does not propagate such definitions to the return node for c. Let $PENDING(c)$ be the set of definitions for memory locations that may be modified by the called procedure at c. Then the data-flow equations for c are:

$$OUT_d(c) = IN_d(c) - PENDING(c) \tag{4}$$
$$OUT_p(c) = IN_p(c) - PENDING(c) \tag{5}$$
$$OUT_k(c) = IN_k(c) - PENDING(c) \tag{6}$$

After computing the IN and OUT sets, the algorithm examines each node n and creates DUAs. For each use of v at n and each reaching definition of v in the $IN(n)$ sets, the algorithm creates a DUA; the algorithm classifies the association by determining the definition type, the use type, and the RD type.

4.2 Interprocedural Phase

In the interprocedural phase, our algorithm first builds the IFG for the program, and then uses the IFG to compute and classify interprocedural DUAs.

The interprocedural flow graph An *interprocedural flow graph* (IFG) for a program contains an IFG subgraph for each procedure P in the program. Each IFG subgraph contains four types of nodes: formal-in,

[7] If GMOD(P) (or GREF(P)) contains a symbolic name, the algorithm creates definitions (or uses) for the actual memory locations that the symbolic name can refer to at call site c.

formal-out, actual-in, and actual-out. Each node is created for a placeholder definition or use added to the CFG for P during the intraprocedural phase. A *formal-in* node is created for each placeholder definition added to the entry node of the CFG. A *formal-out* node is created for each placeholder use added to the exit node of the CFG. An *actual-in* node is created for each placeholder use added to each call node of the CFG. An *actual-out* node is created for each placeholder definition added to each return node of the CFG.

Figure 5 presents the IFG for the program of Figure 1. Symbolic name "?1" (we can view it as `*p`) is used to identify the memory locations whose addresses can be passed into `f1()` through `p`, and thus, "?1" is bound to `x` or `y` at statement 6. The intraprocedural phase added placeholder uses of `x`, `y`, and `g` to the call node for statement 6, and placeholder definitions of `x` and `y` to the return node for statement 6. Therefore, the IFG subgraph for `main()` contain actual-in nodes (nodes 1, 2 and 3) and actual-out nodes (nodes 4 and 5) for those definitions and uses. The IFG subgraph for `f1()` contains formal-in nodes for "?1" and `g` (nodes 7 and 8) and a formal-out node for "?1" (node 9).

Each node in the IFG subgraph for P is associated with a program point in P. A formal-in node is associated with the entry of P. A formal-out node is associated with the exit of P. An actual-in node is associated with the entry of the call statement. An actual-out node is associated with the exit of the call statement. We denote the program point associated with node n as $\mathcal{P}\Downarrow[n]$.

The IFG subgraph contains two types of edges: reaching edges and inter-reaching edges. A *reaching* edge connects a formal-in node or an actual-out node to an actual-in node or a formal-out node. A reaching edge is created to represent a data dependence between a placeholder definition and a placeholder use. Each reaching edge has an RD type associated with it—the RD type of the corresponding DUA. An *inter-reaching* edge is created at a call site and connects an actual-in node to an actual-out node. It summarizes a set of paths in the CFG of the called procedure—the paths through which the definition of a memory location flows from the entry of the procedure to the exit of the procedure. An inter-reaching edge also has an RD type associated with it—the union of the RD types of the CFG paths that it summarizes.

For example, the IFG subgraph for `main()` contains reaching edge (5, 6) because the intraprocedural phase identifies a data dependence with respect to `x` from the return node for statement 6 to the call node for statement 7. The RD type for this data dependence is {green}. The IFG subgraph for `main()` contains inter-reaching edge (2, 5) because there exists a path in the IFG subgraph of `f1()` from a corresponding formal-in node (node 8) to a corresponding formal-out node (node 9). The RD type for this edge is {yellow,green}.

The IFG connects the IFG subgraphs using bind-in and bind-out edges. A *bind-in* edge connects an actual-in node to a formal-in node in the IFG of the called procedure. A *bind-out* edge connects a formal-out node to an actual-out node. We refer to a bind-in edge or a bind-out edge as a *binding* edge. For example, in Figure 5, a bind-in edge connects node 1 to node 7 and a bind-out edge connects node 9 to node 5.

A binding edge usually connects two nodes that are created for the same memory location. In the presence of symbolic names, a binding edge can connect a node created for a symbolic name to another node created for a memory location that is bound to the symbolic name.

Construction of the interprocedural flow graph To build the IFG for a program, our algorithm first constructs the IFG subgraph for each procedure P: it creates nodes and reaching edges in the IFG subgraph using the information computed for P during the intraprocedural phase. Next, the algorithm connects the IFG subgraphs of different procedures using binding edges according to the parameter binding information

at each call site. Finally, the algorithm computes the inter-reaching edges by propagating information from callees to callers.

To compute inter-reaching edges and their RD types, the algorithm processes the procedures in a reverse topological order. For each procedure P, the algorithm traverses the IFG subgraph starting at formal-in nodes and computes reachability to formal-out nodes. If a path exists from a formal-in node e to a formal-out node x, the algorithm creates an inter-reaching edge at call sites that are connected to e and x in the IFG.

To compute the RD type of the path, the algorithm combines the RD types of the edges along the path. If two edges converge at a node, the algorithm uses a union operation to combine their RD types. If an edge follows another edge, the algorithm uses a composition operation to combine their RD types. The *union* of two RD types is a union of the colors that appear in the RD types. A composition operation is based on an ordering among the colors: green $<$ yellow $<$ red. Color a *subsumes* color b if and only if $a \geq b$. The *composition* operation (denoted $\circ$) on two RD types RD_1 and RD_2 performs a pair-wise comparison of the colors that appear in RD_1 and RD_2 and includes the subsuming color for each pair in the composed RD type.

For each procedure P, the algorithm uses a worklist $\mathcal{W}_P$ to compute, for each node n created for a memory v in the IFG subgraph for P, a set of RD types $T[n]$. $T[n]$ contains the RD types of the paths that may reach from the entry of P to $\mathcal{P}\Downarrow[n]$, with respect to v.

$\mathcal{W}_P$ is initialized with the set of reaching edges that leave the formal-in nodes in P's IFG subgraph. Initially, the type set for each formal-in node contains only "green"; the type set for any other node is empty. When our algorithm removes an edge $e = (n_0, n_1)$ from $\mathcal{W}_P$, it updates $T[n_1]$ using the following equation (let $T[e]$ be the set of types contained in the attribute for e):

$$T[n_1] = T[n_1] \cup (T[n_0] \circ RDtype[e]) \tag{7}$$

If $T[n_1]$ is updated and n_1 is not a formal-out node, all the reaching edges or inter-reaching edges that leave n_1 are added to $\mathcal{W}_P$. If $T[n_1]$ is updated and n_1 is a formal-out node created for memory location v, the algorithm updates the RD type of the inter-reaching edges created at each call statement c that invokes P. For each memory location v' that is bound to v at c, our algorithm finds the inter-reaching edges e' created for v at c. Our algorithm updates $RDtype[e']$ using the following equation

$$RDtype[e'] = RDtype[e'] \cup T[n_1] \tag{8}$$

Let Q be the procedure that contains c. If $T[e']$ is updated, then our algorithm adds e' to $\mathcal{W}_Q$. Our algoritm terminates after all the worklists become empty. The expense of the algorithm is linear in the number of edges in the IFG.

Computation of interprocedural DUAs Our algorithm uses the IFG and the DUAs computed during the intraprocedural phase to compute interprocedural DUAs. Our algorithm first identifies in the IFG each path $p_{pu,k,pd} = (n_{pu}, \ldots, n_k, \ldots, n_{pd})$ such that the following conditions hold (with respect to memory location l): (1) n_{pu} is created for a placeholder use pu in procedure P_1 (i.e., n_{pu} is an actual-in or a formal-out node in the IFG subgraph for P_1); (2) n_{pd} is created for a placeholder definition pd in procedure P_2 (i.e., n_{pd} is a formal-in or an actual-out node in the IFG subgraph for P_2); (3) $(n_{pu}, \ldots, n_k)$ is a subpath that contains no bind-in edge; and (4) $(n_k, \ldots, n_{pd})$ is a subpath that contains no bind-out edge. Such a path

indicates that definitions of l in P_1 can reach uses of l in P_2. $p_{pu,k,pd}$ contains an unbalanced-right subpath (n_{pu}, ..., n_k) followed by an unbalanced-left subpath (n_k, ..., n_{pd}); n_k is a node in the IFG subgraph that is reached after the last unmatched bind-out edge in (n_{pu}, ..., n_k). For example, the algorithm identifies path (9, 5, 6, 10) with respect to memory location `x` in the IFG for the example. The algorithm identifies such paths using a two-phase reachability algorithm: in the first phase, the algorithm traverses all edges except bind-in edges, to identify the subpath (n_{pu}, ..., n_k); in the second phase, the algorithm traverses all edges except bind-out edges, to identify the subpath (n_k, ..., n_{pd}).[8]

After identifying $p_{pu,k,pd}$, the algorithm examines actual definitions in P_1 that reach pu, and actual uses in P_2 that are reachable from pd. For each such definition d and use u, the algorithm computes an interprocedural DUA (d, u, l). For example, for the path (9, 5, 6, 10), the algorithm examines definitions of "?1" in `f1()` that reach the placeholder use created at the exit of `f1()`; the algorithm also examines uses of "?2" that are reachable from the placeholder definition created at the entry of `f2()`; the algorithm then computes DUA (12, 16, `x`).

The algorithm classifies (d, u, l) along the four dimensions using the information associated with (d, pu), (pd, u), and the edges in $p_{pu,k,pd}$. The algorithm associates the type of memory location represented by n_k with d and u. If both d and u are created for actual memory locations, the access types for d and u can be determined at the statements where d and u are created. However, if d (or u) is created for a symbolic name, the algorithm determines the access type by examining the bind-out (or bind-in) edges in $p_{pu,k,pd}$. The algorithm computes the RD-type for (d, u, l) by combining the RD-types of (d, pu), (pd, u), and $p_{pu,k,pd}$ using the composition operation. The algorithm determines the path type for (d, u, l) according to the presence of binding edges in $p_{pu,k,pd}$.

Our algorithm computes, for each node n that is created for memory location loc in the IFG, a set of actual definitions that are related to loc and can reach the program point $\mathcal{P}\Downarrow[n]$. For each actual definition def that reaches $\mathcal{P}\Downarrow[n]$, our algorithm computes the following tuple (st, $name$, T_{def}, T_{use}, T_{RD}, T_{path}): st identifies the statement where the definition occurs; $name$, T_{def}, and T_{use} are used to track the changing of names for the memory location and the actual access types of the memory location; T_{RD} and T_{path} are used to compute the RD types and the path types for each def-use association. Each tuple is uniquely identified using st and $name$.

The algorithm first initializes a call node or an exit node n created for loc in P with the set of actual defintions of loc that can reach $\mathcal{P}\Downarrow[n]$, according to the information computed in the intraprocedural phase. For each definition $def = (loc, s)$ that may reach $\mathcal{P}\Downarrow[n]$, our algorithm creates a tuple for n as follows:

- $name$ = name identifying loc in P
- T_{def}=the access type of def
- $T_{use} = unknown$
- T_{RD}=the RD types for def
- $T_{path} = \{SLP\}$

For example, because definition $(?1, 12)$ can reach the exit of `f1()` through a green path, our algorithm creates a tuple $(12, ?1, \mathtt{S}, unknown, \{green\}, \{SLP\})$ at node 9.

Our algorithm uses a two-phase propagation to propagate the actual definitions within the IFG. In the

[8]The system-dependence-graph-based slicing technique uses a similar two-phase graph traversal [8].

first phase, the algorithm propagates information through reaching edges, inter-reaching edges, and bind-out edges, but not bind-in edges. In the second phase, the algorithm propagates information through reaching edges, inter-reaching edges, and bind-in edges, but not bind-out edges. This two phase propagation is a well-known approach for preventing propagation of information through unrealizable paths.

When our algorithm processes a node n during the two-phase propagation, it processes each edge $e = (n, n')$ that leaves n. For each tuple τ computed for n, our algorithm computes a tuple τ' according to the type of e and then adds τ' to n'. If there is already a tuple τ'' in n' with the same identification as τ' (i.e., a tuple τ'' such that $\tau''.st == \tau'.st$ and $\tau''.name == \tau'.name$), our algorithms updates elements in τ'' as follows:

$$\tau''.T_{def} = \texttt{Max}(\tau''.T_{def}, \tau' T_{def}) \tag{9}$$

$$\tau''.T_{use} = \texttt{Max}(\tau''.T_{use}, \tau' T_{use}) \tag{10}$$

$$\tau''.T_{RD} = \tau''.T_{RD} \cup \tau' T_{RD} \tag{11}$$

$$\tau''.T_{path} = \tau''.T_{path}, \tau' T_{path} \tag{12}$$

where `Max` is defined based on the ordering `D` $<$ `S` $<$ `M`.

Tuple τ' is computed as follows:

Case 1: e is a reaching edge or an inter-reaching edge.

$$\tau' = (\tau.st, \tau.name, \tau.T_{def}, \tau.T_{use}, \tau.T_{RD} \circ Attr[e], \tau.T_{path}) \tag{13}$$

Case 2: e is a bind-out edge. Let v be the name that identifies the memory location for n in the callee and v' be the name that identifies the memory location for n' in the caller. If v is a symbolic name that is bound to more than one memory location, then

$$\tau' = (\tau.st, v', \texttt{M}, \tau.T_{use}, \tau.T_{RD}, \tau.T_{path} \circ \{URP\}) \tag{14}$$

Otherwise,

$$\tau' = (\tau.st, v', \tau.T_{def}, \tau.T_{use}, \tau.T_{RD}, \tau.T_{path} \circ \{URP\}) \tag{15}$$

Case 3: e is a bind-in edge. Let v' be the name that identifies the memory location for n' in the callee. If v' is a symbolic name that is bound to more than one memory location, then

$$\tau' = (\tau.st, \tau.name, \tau.T_{def}, M, \tau.T_{RD}, \tau.T_{path} \circ \{ULP\}) \tag{16}$$

Otherwise,

$$\tau' = (\tau.st, \tau.name, \tau.T_{def}, \tau.T_{use}, \tau.T_{RD}, \tau.T_{path} \circ \{ULP\}) \tag{17}$$

Our algorithm uses the interprocedural reaching defintions and the information computed in the intraprocedural phase to compute the def-use associations and their classifications. For each place-holder definition def that is created for loc at the entry of a procedure P, our algorithm computes a set of def-use associations for def when it analyzes P in the intraprocedural phase. For each of those def-use associations $dua = (P.entry, s, loc)$ and for each tuple τ computed for the entry node created for loc in P's IFG subgraph, our algorithm creates an interprocedural def-use association $idua = (\tau.st, s, \tau.name)$. Our algorithm combines the information associated with dua and the information in τ to determine the classification for $idua$ as follows (let T_{RD} be the RD types for dua, and t be the use access type for dua):

- RD types for $idua = T_{RD} \circ \tau.T_{RD}$
- path types for $idua = \tau.T_{path}$
- definition access type for $idua = \tau.T_{def}$
- use access type for $idua = \texttt{Max}(\tau.T_{use}, t)$.

Our algorithm creates interprocedural def-use associations using place-holder definition created at the exit of each call statement in a similar way.

5 Empirical results

To investigate the usefulness of the framework, we implemented a prototype and performed empirical studies with a set of C subjects. We implemented the algorithm for computing interprocedural DUAs using the ARISTOTLE analysis system [6]. To gather data-flow information that is required for alias analysis, we replaced the ARISTOTLE front-end with the PROLANGS Analysis Framework (PAF) [19]. We used PAF to gather control-flow, local data-flow, and symbol-table information; we then used this information to interface with the ARISTOTLE tools. Because PAF does not analyze function pointers, our implementation does not incorporate the effects of function pointers.[9]

We used three alias-analysis algorithms to compute alias information for the empirical studies: Andersen's algorithm [3], which is flow- and context-insensitive and non-parametric; Landi and Ryder's algorithm [9], which is flow- and context-sensitive and non-parametric; and MoPPA [12], which is flow-insensitive, context-sensitive, and parametric. These algorithms vary in the efficiency and the precision with which they compute the alias relations. We used the programs listed in Table 1 as subjects for the empirical studies.

Subject	Description	LOC
T-W-MC	layout generator for cells in circuit design	21385
armenu	Aristotle analysis system interface	11320
bison	Parser generator	5572
dejavu	Regression test selector	3166
flex	Lexical analyser generator	8270
lharc	Compress/extract utility	2550
replace	Search-and-replace utility	551
space	Parser for antenna array description language	6201
tot_info	Statistical information combiner	477
unzip	Compress/extract utility	2906

Table 1: Programs used in the empirical studies.

We investigated the distribution of DUA types in the subject programs using different alias-analysis algorithms.[10] The goal of the study was (1) to investigate the types of DUAs that occur in practice, (2) to evaluate the effects of different alias-analysis algorithms on the distribution of DUA types, and (3) to investigate if the distribution of DUA types can be used to infer characteristics of the programs

We computed the total number of DUA types that occur in all the subjects. Figure 6 presents the results. The figure shows the number of DUA types that occurred in each of the subjects using the three alias-analysis algorithms. The results illustrate that only a limited number of DUA types occur in the considered subjects.

[9] In the subjects, calls involving function pointers have been substituted with calls having the same data-flow effect.

[10] Landi and Ryder's algorithm could not analyze two of the subjects—`T-W-MC` and `bison`—within a reasonable amount of time. Therefore, the results in this section do not contain data for those two subjects using Landi and Ryder's algorithm.

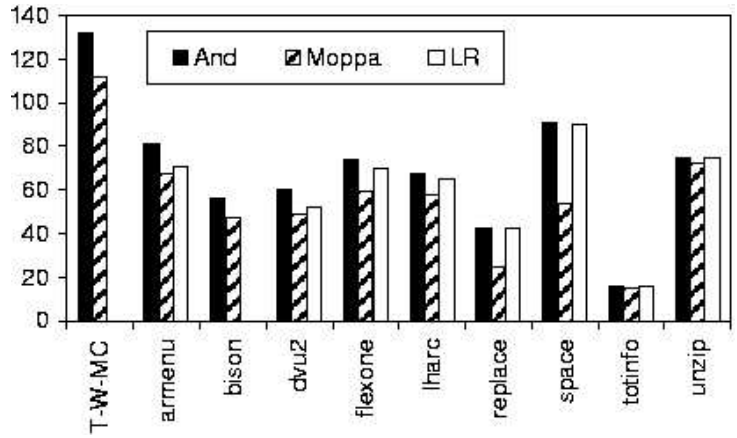

Figure 6: Number of DUA types that appear for each subject.

We expected this result because most of the possible DUA types occur in very restrictive cases—those that involve data flow over complex, recursive program paths—that would occur rarely in practice. The occurrence of only a few types in practice may also be explained by the fact that the usage of language constructs usually follows some specific patterns.

We also determined if the different DUA types occur in significant numbers. We found the same pattern in all subjects and for all alias-analysis algorithms: most of the DUAs fall into a small set of types, and the remaining types account only for a small number of DUAs. To perform this study, we computed, for each subject, the set of DUAs of each type that occurred in the subject. Then we ordered the sets by decreasing cardinality (i.e., the first set in the ordering corresponds to the most-frequently-occurring DUA type, the second set in the ordering corresponds to the second most-frequently-occurring DUA type, and so on). Finally, we grouped the sets incrementally into groups of five, based on the ordering. Thus, the first group of sets includes the five DUA types that occur most frequently in a subject (top 5); the second group of sets includes the 10 most-frequently-occurring DUA types in a subject (top 10); and the third group of sets includes the 15 most-frequently-occurring DUA types in a subject (top 15). By construction, top 5 $\subseteq$ top 10 $\subseteq$ top 15. Finally, we computed the percentage of data DUAs that appear in each group.

	Anderson			MoPPA			LR		
Subject	5	10	15	5	10	15	5	10	15
T-W-MC	81	89	94	74	83	89	–	–	–
armenu	76	86	91	81	89	93	78	87	92
bison	86	95	97	90	98	99	–	–	–
dvu2	79	88	91	82	90	94	84	91	94
flexone	80	88	93	82	90	95	83	91	95
lharc	73	89	93	76	88	93	72	87	94
replace	64	77	86	69	87	97	64	77	86
space	82	89	92	86	93	96	82	90	92
totinfo	95	99	100	95	99	100	95	99	100
unzip	72	84	88	75	86	89	73	84	88

Table 2: Percentage of DUAs that belong to the top 5, top 10, and top 15 groups.

Table 2 shows the results of the grouping. For example, the number 95 in row 3 and column 2 means that, for `bison`, the 10 most-frequently-occurring DUA types accounted for 95% of the total DUAs computed for `bison` (using Anderson's alias-analysis algorithm).

The second goal of our study was to examine the effects of different alias-analysis algorithms on the distribution. Our hypothesis was that a more precise alias-analysis algorithm should cause fewer DUAs, fewer multiple-alias accesses, and fewer yellow paths between definitions and uses. Among the three alias algorithms we used, Landi and Ryder's algorithm is more precise than Anderson's and MoPPA. However, in most cases, the precision of MoPPA is close to—and in some cases exceeds—the precision of Landi and Ryder's algorithm. For the eight of our subjects that we could analyze using all three alias analyses, the fewest DUAs were computed using MoPPA: using MoPPA resulted in 7% fewer DUAs (67,707) than Anderson's algorithm (72,800), and 6.5% fewer DUAs than Landi and Ryder's algorithm (72,470). MoPPA also caused proportionately fewer DUAs with yellow paths between definitions and uses. 53.5% of the DUAs computed using MoPPA had yellow paths between the definitions and the uses; whereas, using Anderson's and Landi and Ryder's algorithm, this proportion was 56.8% and 56.9%, respectively.

Figure 6 shows that the alias information does not affect significantly the total number of DUA types that occur. The total number of types over all subjects is similar for the three alias analysis algorithms: 278 for Anderson's algorithm, 236 for Landi and Ryder's algorithm, and 200 for MoPPA.

The third goal of the study was to investigate if the distribution could provide insights into the characteristics of the programs. To this end, we examined the differences and commonalities among the top 5 groups for different subjects. We found that only two types appear in the top 5 group for all subjects ({scalar, direct, {green}, {IPP}} and {scalar, direct, {green,red}, {IPP}}); one type appears in the top 5 group for four subjects {scalar, direct, {green}, {URLP}}; five types appears in the top 5 group for two subjects; and the remaining 16 types appear in only one top 5 group.

This result is interesting because it suggests that we can characterize a subject based on the types of DUAs that appear predominantly in the subject. To validate this intuition, we looked at the top 5 group for two subjects we are familiar with: `armenu` and `T-W-MC`. The former is a text-based menu interface for the ARISTOTLE analysis system, whose behavior and structure is fairly simple. The latter is used to compute layouts for cells in circuit design, and relies on a dynamic data structure of considerable size to store the graph representing the circuit. We found a strong correlation between our knowledge of the complexity of the subjects and the composition of the top 5 groups for the two subjects. The top 5 group for `armenu` contains only DUA types involving scalar direct definitions and scalar direct uses, which represent the intuitively simplest case along both the memory-location type and the access-type dimensions. Conversely, three of the five DUA types occurring most frequently in `T-W-MC` consist of DUAs involving heap locations and symbolic names used indirectly through a dereference; such DUAs are intuitively more complex to understand than DUAs involving scalar memory locations directly accessed.

6 Summary and Future Work

The overall goal of our research is to improve software-engineering tasks through a better understanding of data dependences. The research involves a number of steps: (1) investigating various properties of DUAs to form a framework for studying and understanding DUAs; (2) applying the framework in various ways to understand DUAs; (3) investigating how to infer overall properties of a program based on the properties of DUAs occurring in the program; and (4) investigating how to leverage information about the characteristics of DUAs to support and improve various software-engineering tasks.

In this paper, we refined our previous classification of DUAs. We presented two new dimensions for classifying DUAs that we did not consider earlier: (1) the type of memory location that is accessed at a definition or use, and (2) the type of path (based on sequences of calls and returns) along which a DUA occurs. Thus, we also extended our earlier work to computing and classifying interprocedural DUAs. We also presented an efficient algorithm for computing and classifying such interprocedural DUAs.

The four dimensions for classifying DUAs can be used as a framework for understanding DUAs. We presented two applications of the framework. In the first application, we obtained a distribution of the types of DUAs in a set of C programs. Our results showed that the distribution of DUAs can be useful in inferring characteristics of a program. Our results also showed that the precision of the alias analysis can affect the types of DUAs: a more precise algorithm results in fewer DUAs and in a higher number of definite, as opposed to possible, DUAs.

In the second application, we discussed how the framework can be used to estimate the complexity of DUAs, and of the statements in which the DUAs occur. We have implemented a visualization tool to view the data-flow complexities of the statements in a program. The tool can be beneficial in isolating segments of the program that have complex data-flow relationships, and thus, can be usefully employed in supporting software-engineering tasks.

In future work, we will investigate other applications of the framework and conduct further empirical studies to evaluate the applications. We will also investigate the usefulness of the complexity metric in estimating the effort required to develop test cases for covering DUAs.

References

[1] H. Agrawal. *Towards Automatic Debugging of Computer Programs.* Ph.D. thesis, Software Engineering Research Centre, Aug. 1991.

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

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

[4] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

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

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

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

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

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

[10] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. of ESEC/FSE '99*, volume 1687 of *LNCS*, pages 199–215. Springer-Verlag, Sept. 1999.

[11] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. Technical report, College of Computing, Georgia Institute of Technology, November 2000.

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

[13] D. Melski and T. Reps. Interprocedural path profiling. Technical Report TR-1382, Computer Sciences Department, University of Wisconsin, Sept. 1998.

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

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

[16] A. Orso, S. Sinha, and M. J. Harrold. Incremental slicing based on data-dependence types. In *Proc. of the Intl. Conf. on Softw. Maint.*, Nov. 2001. (To appear).

[17] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for lang. with pointers. In *Proc. of the Symp. on Testing, Analysis, and Verification*, pages 74–86, Oct. 1991.

[18] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, 1990.

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

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Conf. Record of the 23rd ACM Symp. on Principles of Prog. Lang.*, pages 32–41, Jan. 1996.