Data Flow-Based Test Adequacy Analysis for Languages with Pointers

Thomas J. Ostrand
Siemens Corporate Research, Inc.
755 College Road East
Princeton, NJ 08540
tjo@cadillac.siemens.com

Elaine J. Weyuker ⁕
New York University
251 Mercer Street
New York, NY 10012
weyuker@cs.nyu.edu

Abstract

The data flow adequacy criteria, originally proposed for a simple language and Pascal, have been substantially modified to provide more thorough analysis for code with extensive use of pointers and complex control structures, such as code frequently written in the C language. A prototype tool, TACTIC, has been built to extract def-use associations from C programs, and to determine whether test sets are adequate with respect to the new criteria. TACTIC successfully analyzes data flow of individual functions in C programs with single-level pointer references.

1 Introduction

Test set adequacy assessment based on data flow analysis has been proposed by several authors [5, 7, 9, 12, 13], and at least two systems have been implemented. Rapps and Weyuker [12, 13] introduced a family of test set adequacy criteria known as the *data flow criteria*. Their original theory used a simple language which did not include pointers, structured variables, or subroutine calls.

⁕ This research was supported in part by the National Science Foundation under grant CCR-8920701 and by NASA grant NAG-1238.

The theory was later extended to Pascal [3], and implemented in the ASSET system [2, 4]. Laski has implemented the STAD system [6], to assess *context testing* as described in [7].

Although ASSET handles almost any syntactically correct Pascal program, it uses a simplistic model of pointer and array references, and of interprocedural data flow. In this paper we modify the theory of data flow testing in order to deal specifically with pointer references, while retaining the underlying philosophy of the original theory. We use C as our working language, and describe a prototype version of a data flow test adequacy tool called TACTIC (Test Analysis and Coverage Tool, Intended for C).

The purpose of this paper is to explain the extended theory of data flow testing, and to describe the current version of TACTIC. We do not describe the details of the data flow analysis or aliasing algorithms used in TACTIC.

2 Data Flow-Based Test Adequacy

The motivation behind data flow-based adequacy assessment is to check whether test cases exercise program paths along which data flows. If a variable is assigned a value at some point in a program, and no path using that value is ever exercised, it is unlikely that a faulty assignment would be detected.

The data flow criteria require that a program's test data cause the traversal of subpaths from a variable definition (an occurrence at which the variable is given a value) to either some or all of its uses (occurrences at which the variable's value is referenced). A definition (in the remainder of the paper, this will be called a def) and use connected by such a subpath is a def-use association (the concept will be defined formally later in the paper).

The calculation of defs and uses is complicated by the use of indirect references, aliases, and procedure calls, since the actual storage referenced by a given identifier may not be determinable at compile time. Previous data flow testing work has either ignored pointers and interprocedural flow, or has made simplifying assumptions about their behavior. This can result in either over- or under-estimation (or both) of def-use associations. For example, ASSET assumes that a Pascal pointer reference is distinct from the actual storage it is referencing, and makes no attempt to connect two pointers that might be referencing the same storage. As a result, ASSET can err in two opposite directions in def-use analysis. On the one hand, some potential def-use associations are not detected because the def and the use are not of the same syntactic object, even though they are related through a pointer reference. On the other hand, a path from a def to a use of a given variable might contain an intervening assignment that kills the original def. If the intervening assignment is to an alias of the variable, ASSET would not consider it a def of the variable, and incorrectly declare the existence of an association.

Another type of mistake is made by systems that analyze the semantics of pointers, but perform overly conservative alias analysis. This approach would generate many spurious def-use associations relating defs and uses that, during execution, could never actually represent the same storage.

3 The Data Flow Criteria for Languages with Pointers

3.1 Static Analysis and Def-Use Associations

The goal of data flow-based evaluation of test adequacy remains the same, whether or not the programming language has pointers: to determine if test cases exercise defs, uses, and the paths connecting them. However, static analysis cannot always determine the precise set of def-use associations that exist in a program with pointers. Objects of the C language's fundamental arithmetic types (including **char**, **int**, **float**, and **double**) can be referred to by two types of occurrences:

1. a *program variable* occurrence is any occurrence of a variable whose type is one of the arithmetic types.
2. a *pointer reference* occurrence is a dereferenced expression of type pointer, such as **`*p`** or **`*(p + 4)`**.

In addition to these types of occurrences that refer to storage, we are interested in occurrences of pointer variables themselves:

3. a *pointer variable* occurrence is an occurrence of a variable of type pointer.

Notice that **`*p`** includes both a pointer variable occurrence and a pointer reference occurrence.

Occurrences are classified by the role they play in the program. An occurrence is a **use** if execution of the code containing the occurrence causes the memory location referred to by the occurrence to be read. An occurrence is a **def** if execution of the code containing the occurrence causes the memory location to be written. The constructs permitted in C sometimes lead to defs and uses appearing in unusual places. For example, the parenthesized expression in **`if(a=b+c)`** is simultaneously an assignment statement and a predicate. The occurrence of **`a`** is both a def and a use. In the implicit assignment **`x++`**, **`x`** is both a use and a def. Any occurrence of a variable or pointer in a function argument is a use.

The extensive use of pointers in many programs written in C creates numerous situations in which static analysis cannot determine a unique object that a pointer will reference at run-time. The following definitions distinguish between situations in which there may or may not be ambiguity.

Defn 1: The **alias set** of a pointer expression at a program point is the set of all program variables that the expression could refer to at that point, as determined by static analysis of the code.

Defn 2: A def (or use) occurrence is a **definite def** (respectively, **definite use**) of variable V if static analysis determines that the object being defined (respectively, used) is unambiguously the program variable V. (This includes the case that the occurrence is a pointer reference whose alias set contains a single program variable V.)

Defn 3: A def (or use) occurrence is a **possible def** (respectively, **possible use**) of program variable V if the occurrence is a pointer reference whose alias set contains more than one element, one of which is V. Note that a definite def (respectively, use) is not a possible def (respectively, use).

For convenience, we will sometimes refer to a possible or definite def of an object as a **def** of that object. Similarly, we will sometimes refer to a possible or definite use of an object as a **use** of that object.

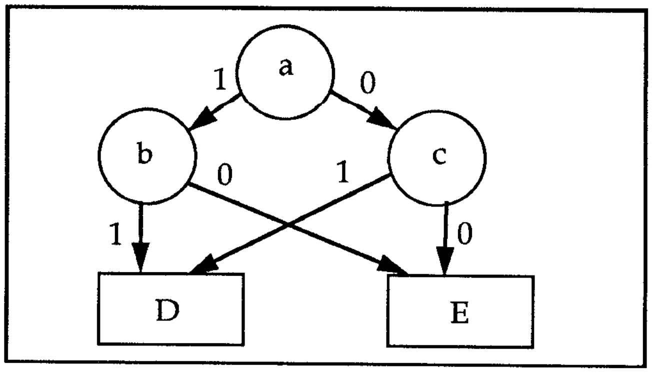
Note two implications of the above definitions: first, by defining alias sets at a program point, we accept some imprecision in alias computation that might be avoidable with path-sensitive alias computation. Second, the ideal concept of an alias set is not necessarily realized by an implemented algorithm for the computation of aliases. If an alias algorithm overestimates alias sets, then the set of possible defs is enlarged, and a variable may appear in an alias set in which it really does not belong. The effect for TACTIC will be evident later in this section, when we define def-use associations.

We will sometimes refer to an occurrence of a dereferenced pointer `*p` as a use or a def of `*p`, when in fact it is the location in memory to which `p` is pointing that is being used or defined. Any occurrence of a pointer variable `p` inside a dereferenced expression is a definite use of `p`, since the value of `p` must be read to calculate the location that `*p` refers to. Thus, `*p` and `*(p+5)` are definite uses of `p`. An assignment to a pointer variable `p`, as in the statement `p = &x;`, is a def of `p` (as it would be if `p` were an ordinary variable), but *not* a def of `*p`. The statement `*p = 17;` contains a def of `*p`, defs of the variables in `p`'s alias set, and a use of `p`. The construct `while (*s++ = *t++)` contains uses of `t` and `s`, defs of `t` and `s`, uses of `*t` and its aliases, defs of `*s` and its aliases, and finally, in the predicate, uses of `*s` and its aliases.

We use a fine-grained form of the program's control flow graph to define def-use associations. The graph consists of one node for each non-control statement, such as an assignment statement, and appropriate nodes to model the flow of each control statement. There is an edge from node n_1 to node n_2 provided it is possible for n_2 to be executed immediately after n_1. Expressions containing certain C constructs must be represented by several nodes to assure that all flows of control are captured. For example, the `if` statement

if (a?b:c) D;
else E;

is represented by the graph shown below. An `if` statement with an ordinary predicate would be represented as a single node with two exiting edges, but the conditional expression predicate requires one node for each of `a`, `b`, and `c`. .



Defn 4: A **path** in a control flow graph is a finite sequence of nodes $(n_1,\ldots,n_j)$, $j \geq 2$, such that there is an edge from n_i to n_{i+1}, $i=1,\ldots,j-1$. A path is **loop-free** if all its nodes are distinct. A path is **simple** if it is either loop-free, or if the first j-1 nodes in the path are all distinct, and n_j is the same node as n_1.

A def-use association is supposed to link a place in the program where a variable is defined with a place where that same def is used. For this to occur, there must be some path from the def location to the use location on which the variable is not redefined. Such a path is a *def-clear path.* To define def-use associations in the presence of pointers, we need a notion of def-clear path that covers the situation where a variable def *might be killed* by an assignment to an alias of the variable. Because of aliasing, static analysis may not be able to determine if a given variable will be defined (at runtime) along a path. We need to distinguish between paths where we *know* that a given variable is not redefined, and paths where there is a possibility that the variable is not redefined.

Recall that a def of ***p** is considered to be a possible def of each of the elements in its alias set, unless the alias set contains a single program variable V. In that case it is a definite def of V.

Defn 5: A path $(n, k_1,\ldots,k_j, m)$, $j \geq 0$ is a **definite def-clear path** from node n to node m w.r.t. a variable V provided there are no defs (definite or possible) of V in the sequence between n and m.

Defn 6: A path $(n, k_1,\ldots,k_j, m)$, $j \geq 0$ is a **possible def-clear path** from node n to node m w.r.t. variable V provided there are no definite defs of V on any node in the sequence between n and m, and there is at least one possible def of V in the sequence between n and m.

The unqualified term *def-clear path* refers to either a possible def-clear path or a definite def-clear path.

We now define a def-use association in general, and then distinguish four types of associations, based on the knowledge obtainable statically about the def, the use, and paths from the def to the use.

Defn 7: A **def-use association** w.r.t. V is a triple (n,m,V) where

a) n contains a (definite or possible) def of V,

b) m contains a (definite or possible) use of V,

c) there is at least one (definite or possible) simple, def-clear path w.r.t. V from n to m.

The following four types of def-use associations form the basis for the definition of our criteria in the next section. They are specializations of Defn 7:

Defn 8: A def-use association is **strong** if n has a definite def of V, m has a definite use of V, and every def-clear path from n to m is definite def-clear w.r.t. V.

Defn 9: A def-use association is **firm** if n has a definite def of V, m has a definite use of V, at least one path from n to m is definite def-clear w.r.t. V, and at least one path from n to m is possible def-clear w.r.t. V.

Defn 10: A def-use association is **weak** if n has a definite def of V, m has a definite use of V, and no path from n to m is definite def-clear w.r.t. V.

Defn 11: A def-use association is **very weak** if either the def or the use or both are possible instead of definite.

We shall sometimes refer to a def-use association simply as an **association.** Notice that the definitions of strong, firm, weak and very weak associations partition the set of associations into four disjoint classes.

In a *strong* association, the def and the use are both of the same variable, there is at least one known def-clear path from the def to the use, and all paths from the def to the use are unambiguously either def-clear or not def-clear with respect to the defined variable.

Firm associations differ from strong ones in that there is at least one path from the def to the use whose def-clearness cannot be ascertained at compile time, in addition to at least one path that is known to be def-clear.

In a *weak* association, *none* of the def-to-use paths can be determined at compile time to be unambiguously def-clear, but the def and use are still known to be of the same variable.

Finally, in a *very weak* association, either or both of the def and the use are indirect references that could correspond to more than one specific variable, and might not be the same.

The concept of an association defined above always relates to a program variable, i.e., to a variable that refers to specific storage. However, in one type of situation, it is also convenient to have a concept of association with respect to a pointer reference ***p**. These cases exist when a def of ***p** is followed by a use of ***p**, the alias sets at the def and use sites are identical, and there is at least one path from the def to the use on which there is no def of any of the aliased variables, of ***p**, or of **p**.

Defn 12: A **def-use association w.r.t. a pointer reference *p** is a triple (n,m,*p), where

a) n contains a def of ***p**,

b) m contains a use of ***p**,

c) the alias set of ***p** at n contains at least two elements, and is equal to the alias set of ***p** at m,

d) there is at least one simple path from n to m that is def-clear w.r.t. every variable V in the alias set of ***p**, and def-clear w.r.t. **p**.

Note that the prohibition of defs of any variable in ***p**'s alias set implies that there can be no defs along the path of a dereferenced pointer ***q** whose alias set at the point of definition includes some element also in ***p**'s alias set.

Pointer reference associations are intended to model data flow via a pointer reference that can be analyzed statically, even though it is not possible to determine which particular program variable is actually being referenced. Note that whenever a pointer reference association exists, there also exist very weak associations w.r.t. each variable in the pointer's alias set.

```
     proc(condition1,condition2)
        int condition1,condition2;
     {
        int x, y, z, *p;
24:          z = 17;
25:          x = 13;
26:          if (condition1) {
28:               p = &y;
29:               *p = z;
             }
             else {
35:               if (condition2)
36:                    p = &x;
37:               else
38:                    p = &z;
40:               *p = 7 + z;
42:               y = 53;
43:               p = &x;
             }
49:          x = x + y + z;
50:          *p = *p + 5;
51:          y = x + y;
```

Figure 1. Example C Program

Figure 1 shows a simple C program that is intended to illustrate the concepts defined above. The program has been constructed to contain examples of most types of def-use associations. A control flow graph for the program appears in Figure 2.

The def of `z` at line 24 participates in three associations: (24,29,z) and (24,40,z) are strong, while (24,49,z) is firm. The latter is firm (rather than strong) because both paths leading from the false branch of `condition1` pass through the statement `40: *p = 7+z;`. Since the aliases of

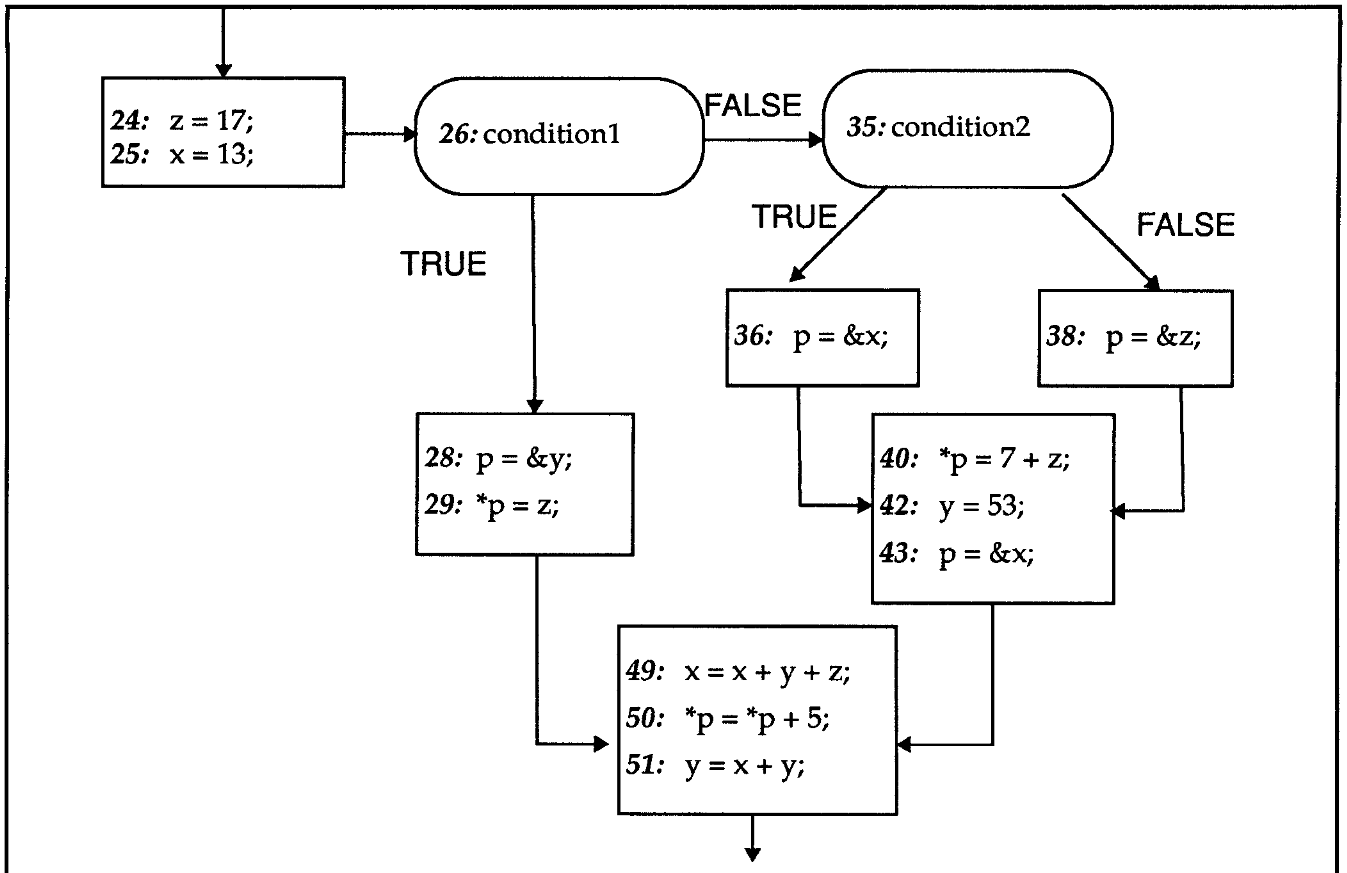


Figure 2. Control Flow Graph for C Program

***p** at this point are {x,z}, that statement is a possible def of **z**, making these paths possible def-clear paths w.r.t. **z**. The association is therefore firm instead of strong. Note that the path segments in question (<24, 25, 26, 35, 36, 40, 42, 43, 49> and <24, 25, 26, 35, 38, 40, 42, 43, 49>) are both possible def-clear, even though the former does not include a redefinition of **z** and the latter does (at statement 40). The reason for this is that alias information is entirely associated with a point rather than a path, and at statement 40 the alias set is {**x**, **z**}.

The def of **y** at line 42 participates in three associations. (42,49,y) is strong, since the path from the def to the use is def-clear with respect to **y**. (42,51,y) is weak, since the only path from the def to the use passes through statement 50, where **y** is included in the alias set of ***p**. Finally, (42,50,y) is a very weak association, since the use in line 50 is a possible use of **y**, as an alias of ***p**.

The latter two associations point up one of the inaccuracies that results from computing aliases at program points. Path sensitive static analysis could show that ***p** can never actually be pointing to **y** along the path <42,43,49,50>. Thus the association (42,51,y) is "strong" in the sense that during execution the def of **y** at line 42 will always reach line 51. Furthermore, (42,50,y) is a spurious association since whenever 50 is reached from 42, **p** is pointing to **x**. (29,51,y) is a different type of spurious association. Here the problem is that **p** *is* pointing to **y** along the path <29,49,50,51>. Although line 29 includes a def of **y**, that def is killed at statement 50, and never reaches 51. Static analysis computes that the aliases of ***p** at line 50 are {x,y}, and hence concludes that <29,49,50,51> is possible def-clear w.r.t. **y**.

A complete list of the def-use associations in the program is shown below.

Strong Associations

(24,29,z) (24,40,z) (29,49,y) (42,49,y)
(28,29,p) (28,50,p) (36,40,p) (38,40,p)
(43,50,p)

Firm Associations

(24,49,z) (25,49,x)

Weak Associations

(29,51,y) (42,51,y) (49,51,x)

Very Weak Associations

(29,50,y) (42,50,y) (50,51,y) (40,49,z)
(40,49,x) (49,50,x) (50,51,x)

3.2 Run-Time Analysis and the Test Adequacy Criteria

We emphasize here that the definitions of def-clear path, def-use association, and the strengths of associations are based entirely on static characteristics of the program. The strength of an association depends on whether the def and the use can be unambiguously determined at compile-time to be a specific variable, and on statically derived information about the paths from def to use. At run-time, the critical issue is to determine which of the statically-defined def-use associations are covered or exercised when the program is executed on a test set.

Defn 13: A def-use association (n,m,V) is said to have been **exercised by a test case** t if

1. execution of t causes control to arrive at node n where a value is assigned to the variable V,
2. control eventually reaches node m, where V is referenced and its value is used in a computation or predicate evaluation, and
3. the value of V is not modified in any statement executed after the value is assigned to V in n until the value is used in m.

A set of def-use associations is said to have been exercised by a test set T provided that every def-use association in the set is exercised by at least one member of T.

To define the criteria, we also need the concept of a test case exercising a def:

Defn 14: A def of variable V at node n is **exercised by test case t** if t exercises some def-use association (n,m,V), i.e., if there exists some node m such that (n,m,V) is a def-use association, and t exercises that association.

We now define the criteria. For all-defs, there will be only one type of criterion, while for all-uses, there will be 4 types, corresponding to the four types of def-use associations.

Defn 15: Test set T satisfies the **all-definitions (all-defs) criterion** for program P if for each def d in the program, d is exercised by some t in T.

Defn 16: Test set T satisfies, respectively, **strong all-uses, firm all-uses, weak all-uses,** or **very weak all-uses** for program P if, respectively, every strong, firm, weak, or very weak def-use association in P is exercised by T.

Note that since the four types of def-use associations are disjoint, satisfaction of the four types of criteria are independent. For example, it is conceivable for a test set to satisfy strong and very-weak all-uses, while not satisfying firm and weak all-uses.

3.3 Evaluating Thoroughness of Tests

Static analysis of a C program produces a list of def-use associations that must be satisfied during test execution for the program to be considered adequately tested. However, all types of associations are not of equal importance for thorough testing. The essential rationale for using data flow coverage to assess the adequacy of test cases is the belief that thorough program testing requires seeing the effect of the value produced by each computation.

In a typical use of TACTIC, the tester would first check to see whether strong all-uses is satisfied. If so, and depending on available resources, the tester would move "down" the criteria checking the satisfaction of the next strongest type of association until either all criteria were satisfied, or testing resources were expended. In this way, the most important associations are given preferential treatment in the sense that if it is not possible to exercise all associations, then the ones that represent definite flows of data must be exercised, while the ones which only represent potential flows may remain unexercised.

Strong and firm associations express a guaranteed relationship between def and use: a quantity is definitely defined, definitely used, and there is a definite (syntactic) def-clear path from the def to the use. Failure to execute a strong or firm association indicates either that all def-clear paths are infeasible, or that the test set really lacks data that could expose a potential fault.

Weak and very weak associations express relationships that correspond to a potential def-use association. Hence failure to execute one of these associations is a less severe criticism of the quality of a test set. There are two possibilities when a test set fails to execute a given weak or very weak association. First, it may be that none of the paths from the def to the use was executed. Second, it may be that one or more of the paths was executed, but that the potential def-use association did not occur, i.e., either the path was not def-clear, or the executed def and the executed use were not of the same variable (in the case of a very weak association). In the first situation, the test set has a real deficiency; the def-use association never had a chance to be executed. In the second situation, although the association has not been exercised, and hence the criterion not satisfied, at least the code connecting the def and use has been exercised.

For the program of Figure 1, inputs are simply pairs of Boolean values. The test set T = {(1,0), (0,0), (0,1)} exercises all paths in the flow graph, and also satisfies strong, firm, and weak all-uses. Although every path in **proc** is feasible, the very weak association (42,50,y) can never be exercised. This association exists because the aliases of ***p** at line 50 are **x** and **y**, making ***p** a possible use of both **x** and **y**. However, when control reaches line 50 on the path <42,43,49,50>, **p** is pointing to **x**, so (42,50,y) is not realized. As already mentioned in Section 3.1, this is a spurious association, generated because aliases are computed at individual lines in the program.

This example shows that a test set that exercises every path in a program does not necessarily exercise every def-use association, in contrast to the original theory, where a test set exercising all paths is guaranteed to exercise every association.

In practice, test set adequacy is assessed by looking at coverage results of all four types of criteria, giving greatest weight to the strong and firm associations, somewhat less to the weak associations, and least to the very weak. If a given test set fails to exercise many (or in some cases any) strong or firm associations, the tester should consider this important evidence that either more testing is needed, or that analysis should be done to determine whether or not the unexecuted strong and firm associations are in fact unexecutable. If only very weak associations remain unexecuted, the tester may decide to stop testing, since the remaining associations may be spurious.

4 The TACTIC System

TACTIC currently runs on SUN 4 workstations. The system consists of three main components: a *pre-execution analyzer* that performs data flow analysis on the source code, calculates def-use associations, and instruments the tested program; an *execution monitor* that traces the program's execution, records paths, and checks off exercised def-use associations; and the external *user interface.*

4.1 Pre-Execution Analysis

Pre-execution analysis starts with a C source file, and produces two results: def-use association information based on static data flow analysis of the code, and a traceable object module. Three separate processing steps are applied to the source code. Initially source-to-source transformations are performed that replace the original C code with code that is more amenable to static analysis. The output of this step is used by both of the following steps. The transformations facilitate the static analysis, and permit more accurate tracing and reporting of defs and uses in the executed program. They are implemented in the Program Transformation Toolkit of the Maintainer's Assistant [1] developed at Siemens research laboratories. The transformations first convert the source program to an abstract syntax tree representation, then transform the tree, and finally convert the transformed tree back into source code.

The second processing step is the static analysis that identifies defs, uses, and def-use associations. The algorithms for calculating def-use associations perform program-point-specific alias computations, for one level of pointer indirection. The current version of TACTIC (July 1991) does not calculate interprocedural associations. We intend to add this capability shortly, based on the algorithms for aliasing and reaching definitions described in [11].

Instrumentation is accomplished by compiling the transformed source program with the GNU-Emacs C compiler (GCC), using the AE option [8]. This compilation produces a traceable object module with hooks that allow the AE tool to recover memory addresses referenced during execution. The addresses are used by the TACTIC run-time monitor to detect the exercise of def-use associations.

4.2 Execution Monitoring

Because TACTIC monitors indirect reference def-use associations, its run-time monitoring is quite different from that done by the ASSET system. In ASSET, covering a def-use association means that one of a set of path segments is exercised. Hence, an association can be verified by instrumenting a program's basic blocks with probes, and noting at run-time whether or not the blocks that make up one of the necessary segments are executed. However, for all the new pointer criteria except strong all-uses, such block instrumentation is not sufficient. The firm and weak all-uses criteria can be satisfied by the execution of a path between the def and the use that is possible def-clear, provided the def is not killed on the path. Since some executions of a path segment may kill the def, while others do not, merely checking whether a path segment is executed is not enough. It is also necessary to check whether the def remains alive at the use site. For very weak associations, it is also necessary to check at run-time whether the def and the use actually represent the same variable.

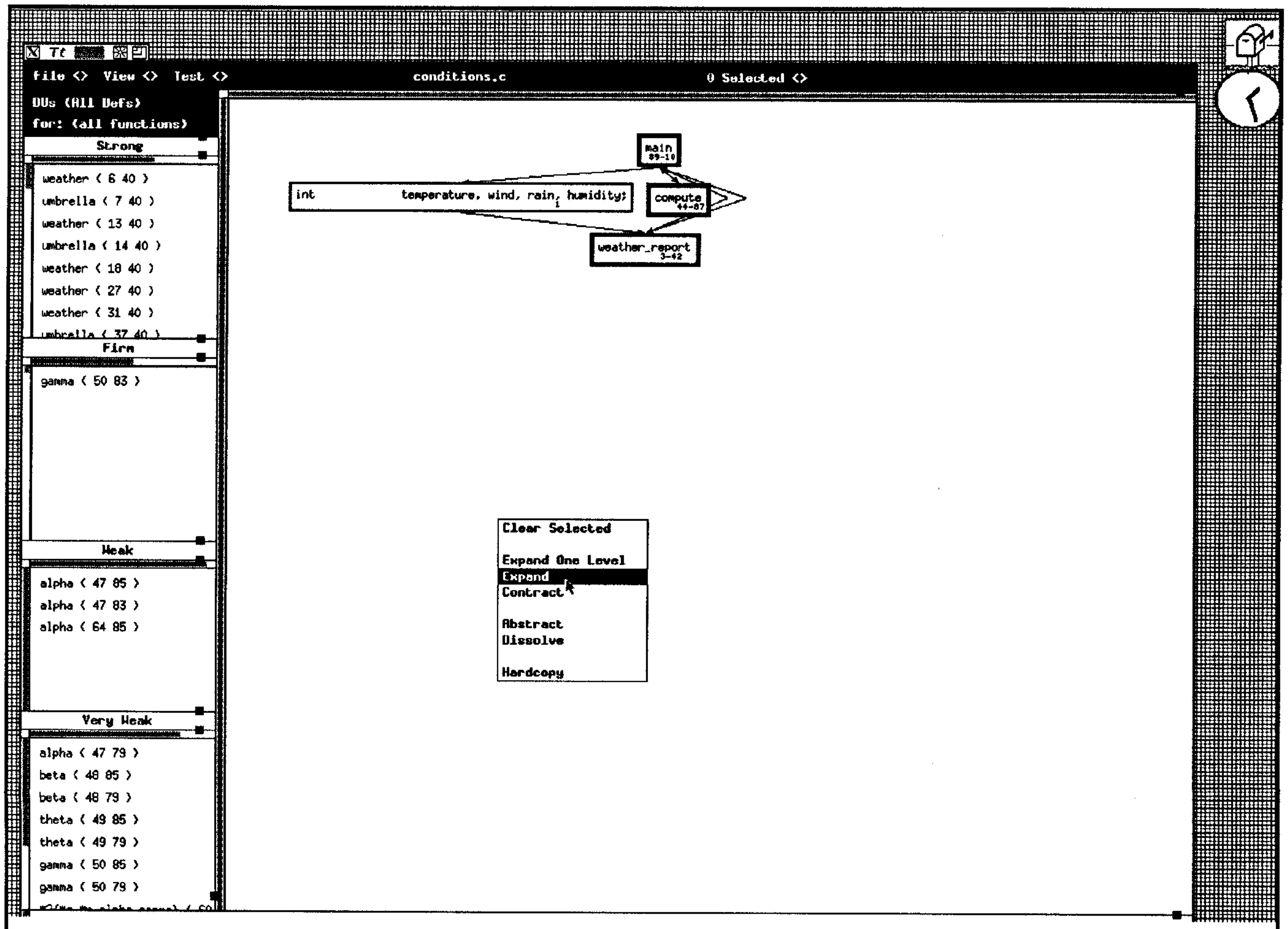
During execution, the GCC-compiled program is traced under control of AE. References to memory generated by a read or write operation are monitored by AE, and mapped into references to source program variables or pointer references, and their occurrences on specific lines of the program. When a write and a read operation are found to refer to the same memory object, and there is no intervening write, then a def-use association has been exercised, and is marked off in the list of static associations.

In addition to dynamic checking of indirect references for *inter-block* associations, TACTIC also analyzes and checks for *intra-block* def-use associations. In ASSET, intra-block associations are not analyzed, since all the code within a block is executed whenever the block is entered, and the meaning of indirect references is the same at run-time as at compile-time. However, with the analysis of pointer aliases, situations can occur in which a block is executed but a particular association within the block is not exercised. For example, consider the program of Figure 1. At line 50, the alias set of ***p** is {x,y}. The def of ***p** generates the two very weak associations (50,51,x) and (50,51,y). Since at line 50, **p** will be pointing at **x** for some inputs, and at **y** for others, a given test case will exercise either one or the other of the associations, even though both statements 50 and 51 are always executed. A related situation can occur for a weak association if it is based on exactly one possible def-clear path that is contained entirely within a single block. Such a path contains a possible def of the variable of the association; if the def actually redefines the variable during an execution of the block, then that execution does not exercise the association. This is another example demonstrating that a test set exercising all paths does not necessarily exercise all associations.

4.3 User Interface

The TACTIC interface is designed to present maximum information about the progress of program testing, using the data flow criteria as adequacy criteria. It provides a simple interface for executing the test program either on individual test cases supplied from the keyboard, or on test cases from a file.

Since data flow adequacy is defined in terms of execution of certain control flow paths, a graphical view of the program is provided to help the tester visualize the extent of coverage achieved, determine the locations of unexercised def-use associations, and derive test cases to exercise them. Figure 3 shows the TACTIC screen after a file called **`conditions.c`**, containing the three functions **`main`**, **`compute`**, and **`weather-report`**, has been loaded by TACTIC. The displayed graph shows the program's top level, namely the three procedures and the control flow that connects them. At the left of the screen are four scrollable windows showing, respectively, strong, firm, weak, and very weak def-use associations. This information can be shown for the entire program currently loaded (as in Figure 3), or can be restricted to associations from just one function of the program.



```
weather ( 6 40 )
umbrella ( 7 40 )
weather ( 13 40 )
umbrella ( 14 40 )
weather ( 18 40 )
weather ( 27 40 )
weather ( 31 40 )
umbrella ( 37 40 )
```

```
gamma ( 50 83 )
```

```
alpha ( 47 85 )
alpha ( 47 83 )
alpha ( 64 85 )
```

```
alpha ( 47 79 )
beta ( 48 85 )
beta ( 48 79 )
theta ( 49 85 )
theta ( 49 79 )
gamma ( 50 85 )
gamma ( 50 79 )
```

Figure 3. TACTIC Screen: Initial Program Load

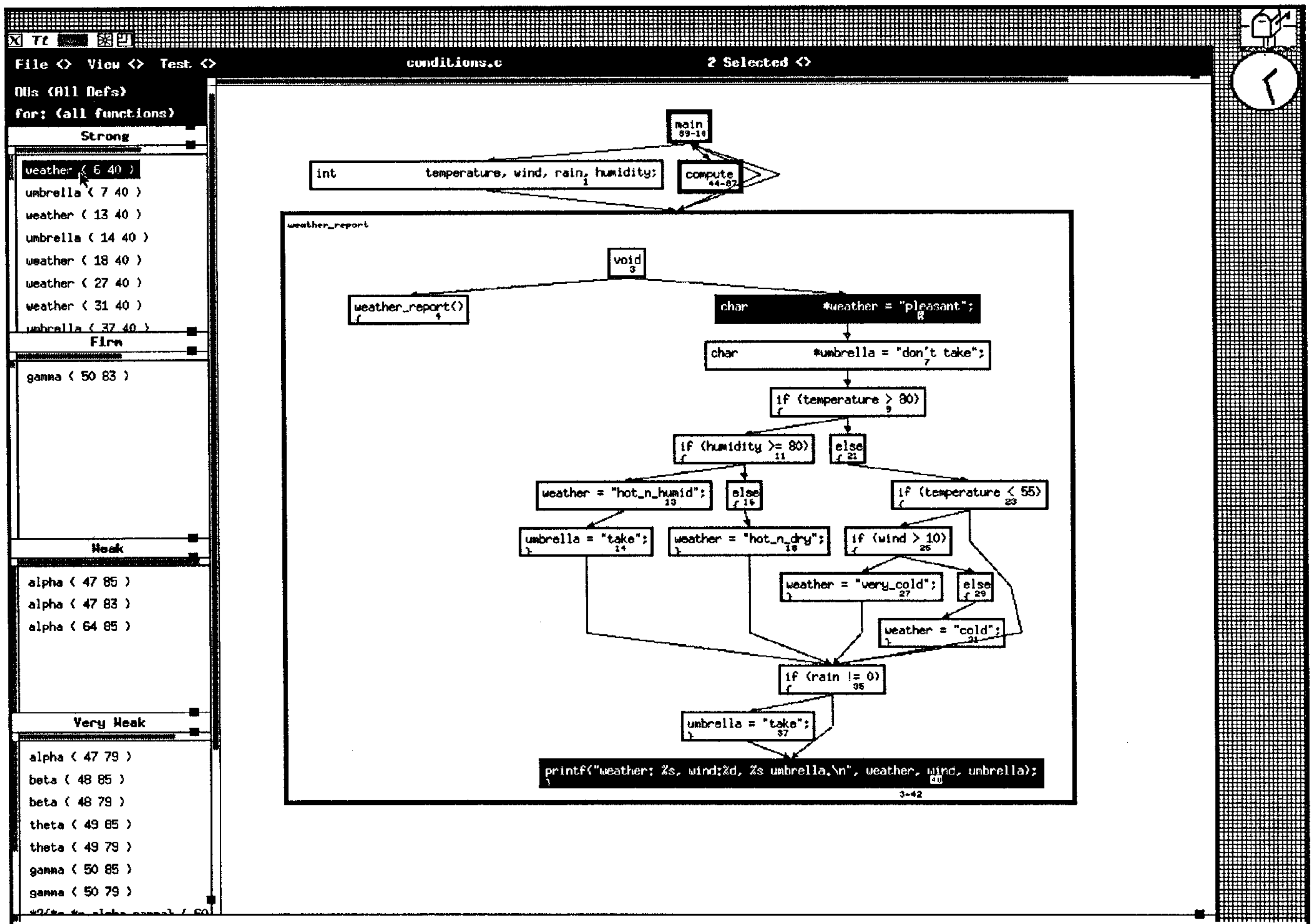


Figure 4. TACTIC screen: Expanded Control Flow Graph

The pop-up menu below the control flow graph allows the user to control the level of detail of the displayed graph. Selecting a single node and applying *Expand* "opens up" the selected node, by replacing it with the complete lowest level control flow it contains. An example of this is shown in Figure 4, in which the **weather-report** node has been expanded. The *Expand-one-level* operation replaces a node with all its top-level control structures. It is also possible to remove detail from the displayed graph; applying the *Contract* operation to a set of nodes that previously resulted from an *Expand* collects the selected nodes back into a single node.

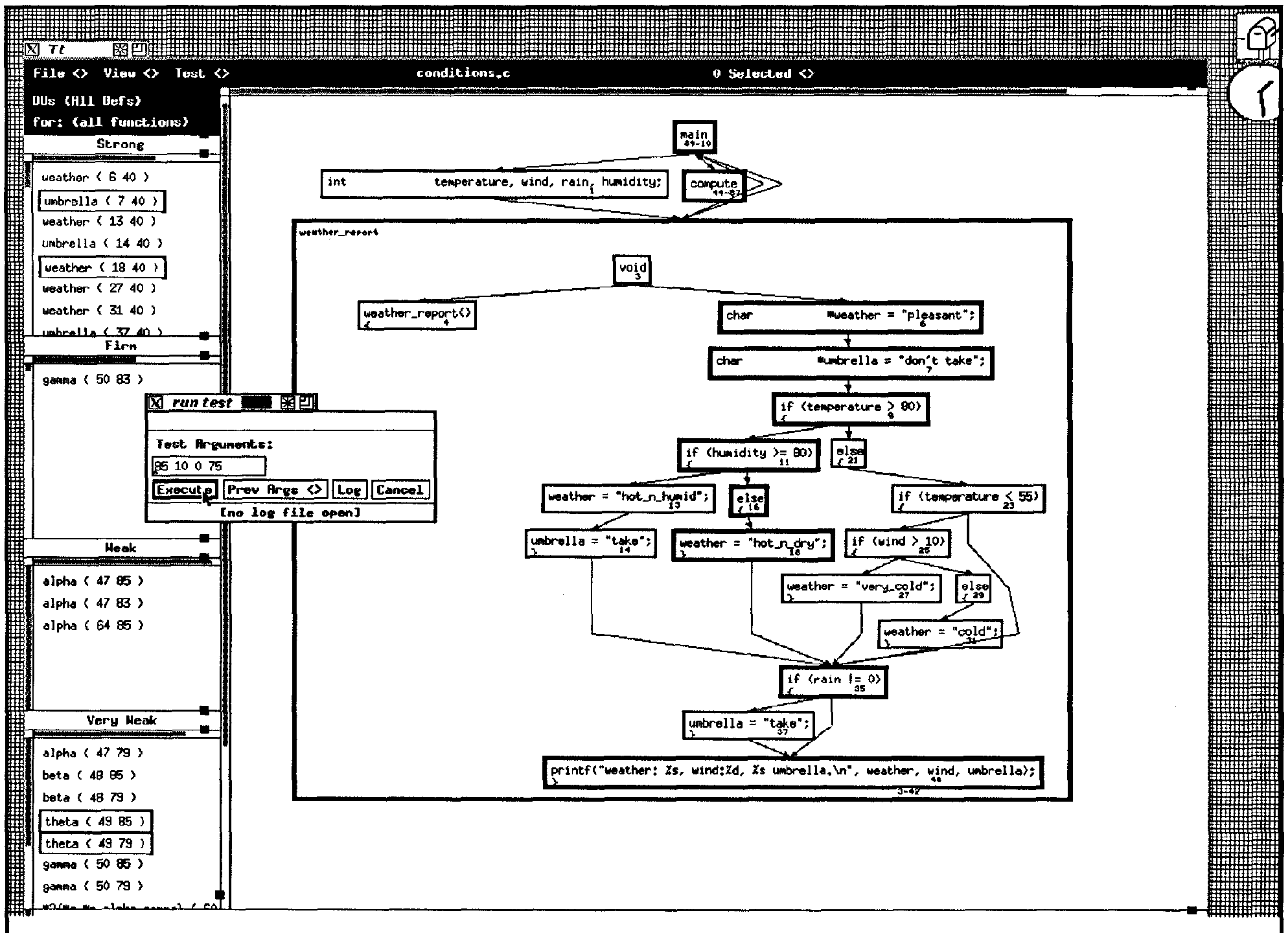
Figure 4 also shows how TACTIC maps from the list of def-use associations to the control flow graph. The user has selected the def-use association *(weather 6 40)* from the Strong window, and asked TACTIC to locate that association in the graph. As a result, the boxes containing the code that defines and uses *weather* are highlighted. If the user requests a display of a def-use association when a control flow graph is too large to fit completely on the screen, TACTIC attempts to move the graph so that the def and use nodes are both visible.

In addition to displaying static properties of the tested code, TACTIC provides an interactive interface for running test cases and displaying the results. Figure 5 shows the ***run-test*** pop-up menu just after the test case (85,10,0,75) has been supplied to the program. The test argument box allows the user to type inputs that are read from "standard in". When the test case is executed, the control path that it follows is highlighted in the graph, at whatever level of abstraction is currently displayed. The user can at any time request that the graph be cleared of all displayed paths, or that the path followed by any previously executed test case be highlighted. It is also possible to display the cumulative path coverage of test cases, thus giving a quick visual overview of the extent of code coverage achieved by the tests.

Figure 5 also illustrates how TACTIC displays def-use coverage information. When the test case runs, def-use associations that are exercised are boxed. When the criterion in effect is all-defs, the exercise of *any* def-use association involving a given def results in the boxing of *all* associations that involve that def. A pull-down menu from the DUs button (in the upper left of the screen) gives the user the options of removing exercised def-use associations from the windows, purging the DU state (i.e., removing the effect of any previously executed test cases), saving the DU state, or setting the criterion to either all-uses or all-defs. If the criterion is changed after some tests have been run, TACTIC updates the def-use association windows to accurately reflect the coverage according to the new criterion.

5 Summary

The data flow adequacy criteria, originally proposed for a simple language and Pascal, have been substantially modified to analyze languages like C with extensive use of pointers. A prototype tool, TACTIC, has been built to extract def-use associations from C programs, and to determine whether test sets are adequate with respect to the new criteria. TACTIC successfully analyzes data



```
weather ( 6 40 )
umbrella ( 7 40 )
weather ( 13 40 )
umbrella ( 14 40 )
weather ( 18 40 )
weather ( 27 40 )
weather ( 31 40 )
umbrella ( 37 40 )

gamma ( 50 83 )

alpha ( 47 85 )
alpha ( 47 83 )
alpha ( 64 85 )

alpha ( 47 79 )
beta ( 48 85 )
beta ( 48 79 )
theta ( 49 85 )
theta ( 49 79 )
gamma ( 50 85 )
gamma ( 50 79 )
```

```
main
int temperature, wind, rain, humidity;
compute
void
weather_report()
char *weather = "pleasant";
char *umbrella = "don't take";
if (temperature > 80)
if (humidity >= 80)
else
weather = "hot_n_humid";
else
if (temperature < 55)
umbrella = "take";
weather = "hot_n_dry";
if (wind > 10)
weather = "very_cold";
else
weather = "cold";
if (rain != 0)
umbrella = "take";
printf("weather: %s, wind:%d, %s umbrella.\n", weather, wind, umbrella);
```

Figure 5. TACTIC screen: Run Test Case

flow of individual functions in C programs with single-level pointer references. We are currently working on extending both the theory and the TACTIC tool to handle arrays and structures, and to deal with interprocedural data flow in the presence of pointers.

6 Acknowledgements

Tarak Goradia and Bingchiang Jeng designed and built the first TACTIC implementation, and also made significant contributions to formulating the theory of data flow-based testing in the presence of pointers. Arup Acharya designed and wrote the data flow analysis code. Monica Hutchins designed and implemented the window interface and the graph processing algorithms. We had many discussions with Hemant Pande on the concepts of aliasing, reaching definitions, and live variables, and their relation to test adequacy analysis. Michael Platoff and Mike Wagner reviewed the system and the theory at various stages, and gave us a number of useful suggestions.

References

1 J. Camaratta, M. Platoff, and M. Wagner, "An Integrated Program Representation and Toolkit for the Maintenance of C Programs", Proc. IEEE Conf. on Software Maintenance-1991, (October 1991).

2 P.G. Frankl and E.J. Weyuker, "A Dataflow Testing Tool", Proc. IEEE Softfair II, San Francisco, (December 1985).

3 P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", IEEE Trans. Softw. Eng. SE-14, (October 1988).

4 P.G. Frankl, S.N. Weiss, and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", Proc. IEEE Conf. Software Tools, New York, (April 1985).

5 P.M. Herman, "A Data Analysis Approach to Program Testing", Australian Computer J. 8, (November 1976), 92-96.

6 J. Laski, "Data FLow Testing in STAD", J. Systems Software 12 (1990), 3-14.

7 J.W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy", IEEE Trans. Softw. Eng. SE-9, no 3, (May 1983), 347-354.

8 J. R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs", Software--Practice and Experience 20, (December 1990), 1241-1258.

9 S. Ntafos, "An Evaluation of Required Element Testing Strategies", Proc. Seventh Int. Conf. Software Engineering, (March 1984), 250-256.

10 T. Ostrand, "Data-Flow Testing with Pointers and Function Calls", Proc. Pacific Northwest Software Quality Conf., Portland, OR (Oct. 1990), 218-227.

11 H. D. Pande, B. G. Ryder, and W. A. Landi, "Interprocedural Def-Use Associations in C Programs", Proc. ACM SIGSOFT Symposium on Software Testing, Analysis, and Verification, VIctoria, B.C. (October 1991).

12 S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection", Proc. Sixth Int. Conf. Software Engineering, Tokyo, (September 1982).

13 S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Trans. Softw. Eng. SE-11, (April 1985).