Data Flow Testing

Neelam Gupta
The University of Arizona, Tucson, Arizona
and
Rajiv Gupta
The University of Arizona, Tucson, Arizona

Software testing techniques are designed to execute a program on a set of test cases that provide sufficient coverage under some specific well-defined test coverage criterion. One form of testing is data flow testing which is aimed at exercising statically detected definition-use associations in a program. A data flow based test coverage criterion characterizes a minimal set of definition-use associations that must be exercised by the test cases before the testing of the program is considered to be complete. In this chapter we describe various data flow based test coverage criteria and describe some commonly used strategies for testing a program under these criteria. In order to carry out testing, test cases are required. We also discuss the approaches for generating test cases including their manual generation as well as automatic generation.

1. INTRODUCTION

While the employment of systematic design and development practices results in increasingly reliable software, some errors are still likely to be present in the software. The goal of testing is to expose hidden errors by exercising the software on a set of test cases. In its simplest form, a test case consists of program inputs and corresponding expected outputs. Once the software has successfully gone through the testing phase, we have a greater degree of confidence in the software's reliability.

Software testing is very labor intensive and hence also expensive. It can account for 50% of the total cost of software development [6]. Therefore tools that automate one or more aspects of testing can greatly help in managing the overall cost of testing. Testing techniques can be broadly classified into two categories, *functional* and *structural*. Functional testing is concerned with functionality rather than implementation of the program. Therefore it involves exercising different input output conditions. Structural testing is concerned with testing the implementation of the program by exercising different programming structures used by the program.

The primary focus of this chapter is structural testing. Before we discuss structural testing in greater detail, let us identify the key aspects of a structural testing strategy.

. *What is the form of test requirements?*
We must identify specific program entities in terms of which the *test requirements* of a program can be stated. When the program is run on a test case, we can analyze the program's execution to determine the set of test requirements that are exercised by the test case. Some examples of test requirements are: program statements, program paths, definition-use associations, and definition-use paths.

. *How much testing is enough?*
We need to determine when the testing process has been completed. For this pur-

pose we define *test coverage criteria.* A test coverage criterion specifies a minimal set of test requirements that must be collectively exercised by the test cases on which the program is executed during the testing process. Depending upon the criterion, a minimal set of test requirements may or may not be unique. The test criterion also helps us guide the testing process. At any point during the testing of a program, our goal is to run the program on test cases that exercise test requirements that are yet to be covered by any of test cases that have already been executed.

. *What is the testing strategy?*
Typically first *unit testing* is employed to test all the modules in the program individually and then *integration testing* is employed to test the interfaces among the modules. There are different ways to organize the integration testing process. While initially a program is *fully* tested, during the maintenance stages of a program, *regression testing* techniques are employed to exercise only the test requirements that are impacted by the program changes. Finally the generation of test cases required to test a given set of test requirements can be carried out *manually* or some *automatic* techniques can be employed.

There are three types of structural testing techniques: *control flow based* testing, *data flow based* testing, and *mutation* testing. Control flow based test coverage criteria express testing requirements in terms of the nodes, edges, or paths in the program control flow graph. The data flow based test coverage criteria express testing in terms of definition-use associations present in the program. Mutation testing begins by creating mutants of the original program which are obtained by making simple changes to the original program. The changes made to the original program correspond to most likely errors that could be present. The goal of testing is to execute the original program and its mutants on test cases that distinguish them from each other. While the focus of this chapter is on *data flow testing*, we also briefly comment upon *control flow testing* for comparison purposes.

The steps of the data flow testing are summarized in Fig. 1. A test coverage criterion is chosen and test cases are generated to adequately test the program under the selected criterion. When the test coverage criterion is satisfied, testing terminates. If the program output differs from the expected output for some test case, the program is debugged and modified and then testing resumes.

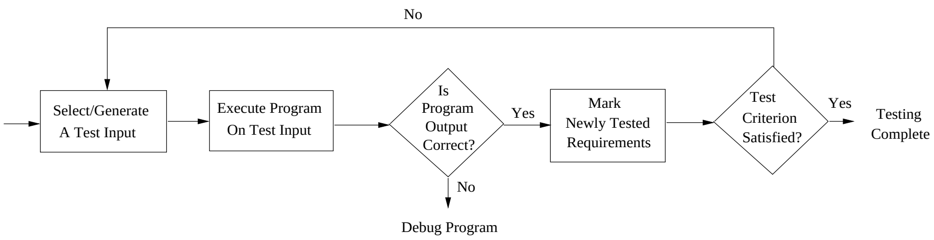


Fig. 1. The Testing Process.

The remainder of this chapter is organized as follows. In section 2 we introduce the various test coverage criteria aimed at exercising the program's control flow and data flow characteristics. In section 3 we discuss commonly used strategies for complete testing of a program under a data flow based testing criterion. In section 4 we describe regression testing techniques that are employed after a program change is made during program maintenance. The test input generation problem is considered in section 5. We briefly describe some approaches to automating test case generation. Concluding remarks are given in section 6.

2. TEST COVERAGE CRITERIA

A test coverage criterion is used to determine whether a program has been adequately tested. It specifies the minimal set of program entities that must be collectively exercised by the test cases on which the program is executed during the testing process. Given a set of test cases, by executing the program on those test cases, and examining the paths followed during the program's executions, we can determine whether or not a given test coverage criterion is satisfied.

2.1 Control Flow Coverage Criteria

A simple form of structural test coverage criteria are based upon the characteristics of the program control flow graph. The test coverage criteria included in this category are:

. *All-nodes* test coverage criterion requires that each node in the control flow graph be executed by some test case. Therefore it is also called *statement testing*.

. *All-edges* test coverage criterion requires that each edge in the control flow graph be traversed during some program execution. This form of testing is also called *branch testing* because each branch outcome is exercised under this criterion. Various variations on branch testing have been proposed to rigorously test the branch predicates [13]. The *all-edges* criterion subsumes the *all-nodes* criterion because if all edges in the flow graph are exercised by the test cases, then it is guaranteed that *all-nodes* are also exercised by the test cases; however, the reverse is not true.

. *All-paths* criterion requires that every *complete path* (i.e., a path from the *entry* node to the *exit* node of the flow graph) in the program be tested. This form of testing is also called *path testing*. The *all-paths* criterion subsumes the *all-edges* test criterion. For a program with loops, there may be infinite complete paths. Therefore proposals have been made to limit the testing to a finite set of paths. A popular technique that falls in this category is McCabe's basis path testing technique. This technique requires a set of paths to be tested that cover all the branches in the program. Therefore, McCabe's basis path testing technique provides exactly the same coverage as the *all-edges* test coverage criterion.

From the above descriptions of the three criterion it is clear that the degree of testing performed differs from one test criterion to another. The *all-nodes* criterion performs the least amount of testing and is not considered effective in exposing faults because it does not sufficiently exercise interactions between statements. The *all-paths* criterion is very stringent but it is impractical for realistic programs.

2.2 Data Flow Coverage Criteria

Data flow testing was introduced to close the gap between *all-edges* and *all-paths* criteria [14; 35; 34]. Data flow testing criteria are aimed at exercising definition-use associations in the program. The exercising of a definition-use association can be viewed as requiring traversal of a selected subpath which originates at the definition, terminates at the use, and is responsible for establishing the definition-use association. Before introducing the data flow based testing criteria, we give following relevant definitions.

. Each occurrence of a variable on the left hand side of an assignment is a *definition* of the variable as it updates the value of the variable.

. Each occurrence of a variable on the right hand side of an assignment is called a *computation use* or a *c-use*.

. Each occurrence of a variable in a boolean predicate results in a pair of *predicate uses* or *p-uses* corresponding to true and false evaluations of the predicate.

. A path $(i, n_1, n_2, \cdots, n_m, j)$ is a *definition-clear* path from the exit of i to entry of j if n_1 through n_m do not contain a definition of x.

. Given a definition of x in node n_d and a c-use of x in node n_{c-use}, the presence of a *definition-clear* path for x from n_d to n_{c-use} establishes the *definition-c-use association* (n_d, n_{c-use}, x).

. Given a definition of x in node n_d and a p-use of x in node n_{p-use}, the presence of a *definition-clear* path for x from n_d to n_{p-use} establishes a pair of *definition-p-use associations* $(n_d, (n_{p-use}, t), x)$ and $(n_d, (n_{p-use}, f), x)$ corresponding to true and false evaluations of the predicate in n_{p-use} respectively.

. The paths through the program along which definition-use associations are established are called *du-paths.* A path $(n_1, \cdots, n_j, n_k)$ is a *du-path* for variable x if n_1 contains a *definition* of x and either n_k has a *c-use* of x and $(n_1, \cdots, n_j, n_k)$ is *definition-clear simple path* for x (i.e., a definition-clear path with all nodes, except possibly the first and last, being distinct) or n_j is a *p-use* of x and $(n_1, \cdots, n_j)$ is a *definition-clear loop-free path* for x (i.e., a definition-clear path in which all nodes are distinct).

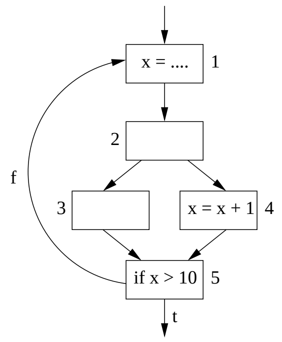


Fig. 2. Illustration of Definitions.

The example in Fig. 2 contains two definitions of x in nodes 1 and 4, one *c-use* of x in node 4, and a pair of *p-uses* of x due to its use by the predicate in node 5. The paths $(1, 2, 3, 5)$ and $(1, 2, 4)$ are definition clear paths for x. The *du-path* $(1, 2, 4)$ establishes the *definition-c-use* association $(1, 4, x)$ while the *du-path* $(1, 2, 3, 5)$ establishes *definition-p-use* associations $(1, (5, t), x)$ and $(1, (5, f), x)$.

Given a set of test cases, let P be the set of complete paths exercised by the program executions for these test cases. For each of the data flow based test coverage criterion, the conditions that P must meet for the test criterion to be satisfied are given below.

. *All-defs* is satisfied if P includes a definition-clear path from every definition to some corresponding use (c-use or p-use).

. *All-c-uses* is satisfied if P includes a definition-clear path from every definition to all of its corresponding c-uses.

. *All-c-uses/some-p-uses* is satisfied if P includes a definition-clear path from every definition to all of its corresponding c-uses. In addition, if a definition has no c-use, then P must include a definition-clear path to some p-use.

. *All-p-uses* is satisfied if P includes a definition-clear path from every definition to all of its corresponding p-uses.

. *All-p-uses/some-c-uses* is satisfied if P includes a definition-clear path from every definition to all of its corresponding p-uses. In addition, if a definition has no p-use, then P must include a definition-clear path to some c-use.

. *All-uses* is satisfied if P includes a definition-clear path from every definition to each of its uses including both c-uses and p-uses.

. *All-du-paths* is satisfied if P includes all du-paths for each definition. Therefore if there are multiple paths between a given definition and a use, they must all be included.

. *Oi-all-uses* is satisfied if P includes a definition-clear path from every definition to each of its uses including both c-uses and p-uses. Moreover each use is considered tested only if it is *output influencing*, that is, it directly or indirectly influences the computation of some program output during the program run [9].

From the above definitions of different data flow based criteria it is clear that each criterion identifies a minimal set of definition-use associations that must be exercised by a set of test cases to satisfy the criterion. However, this minimal set is not unique. As was the case with control flow based testing criteria, subsumption relationships also exist between certain data flow based criteria. In Fig. 3 these subsumption relationships are identified. We have also included the control flow based test coverage criteria in this hierarchy for completeness. Dashed lines are used to distinguish them from the data flow based test coverage criteria.

Let us consider the example in Fig. 4 to illustrate the test criteria. In addition to the control flow graph, two test inputs and their corresponding complete paths are given in the figure. The two complete paths cover all nodes and edges in the flow graph and therefore satisfy the *all-nodes* and *all-edges* criteria. Now let us consider the various data flow based test coverage criteria. The table in Fig. 4 lists all of the definition-use associations present in the program. For each of the data flow based criterion, excluding the *all-du-paths* and *oi-all-uses* criteria, a minimal set of

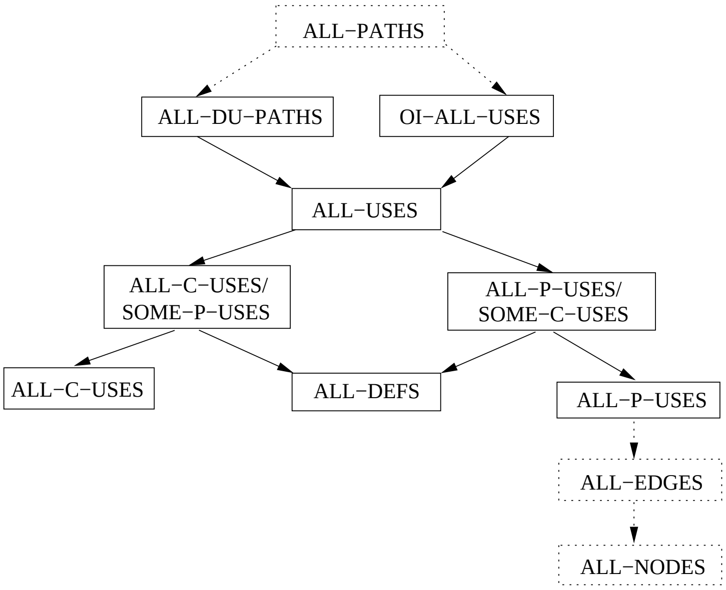


Fig. 3. Test Coverage Criteria Hierarchy.

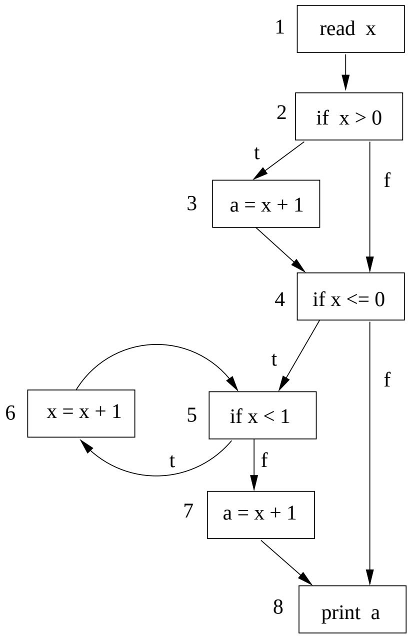
associations are identified. If a given set of complete paths exercise the specified associations for a test criterion, then they satisfy the test criterion. In this example, all of the specified test criteria are satisfied by the two complete paths corresponding to the two program inputs. The *all-du-paths* criteria is also specified by the two complete paths as in this example there is only a single du-path corresponding to each definition-use association.

From the example in Fig. 4, we can make the following additional observations.

. If a more stringent test coverage criterion is chosen, the number of definition-use associations that must be tested increases. For example, to satisfy the *all-defs* criterion we need to test only four associations while to satisfy the *all-uses* criterion we must test all associations.

. The minimal set of associations that must be tested to satisfy a given criterion is not always unique. For example, to satisfy the *all-defs* criteria any one of the first nine associations can be selected to test the definition of x in node 1. The test requirements and/or test cases can be prioritized to exploit the above characteristic. In particular, priorities can be used to reduce the testing effort [19] or maximize the rate of fault detection [17; 36].

. Testing greater number of associations does not always require execution of greater number of test cases. In our example, while the minimal number of definition-use associations that must be tested to satisfy various criteria varies greatly, all coverage criteria are satisfied by the two test cases. In fact it has been shown that the *all-uses* criteria is quite practical as relatively few test cases are required to satisfy this criterion [41].

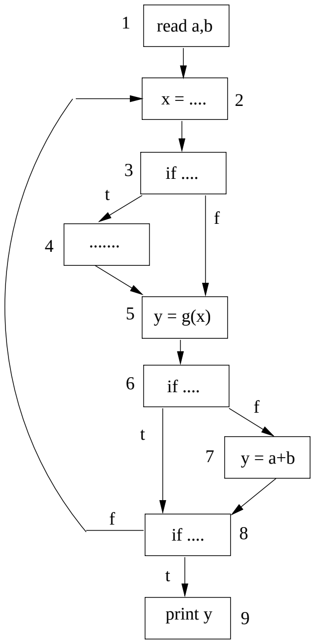


Test Input	Complete Path
x = 1	P_1 : 1-2-3-4-8
x = -1	P_2 : 1-2-4-5-6-5-6-5-7-8

Criteria $\rightarrow$ / Association $\downarrow$	all-defs	all-c-uses	all-p-uses	all-c-uses/ some-p-uses	all-p-uses/ some-c-uses	all-uses
(1,(2,t),x)	$\surd$		$\surd$		$\surd$	$\surd$
(1,(2,f),x)			$\surd$		$\surd$	$\surd$
(1,3,x)		$\surd$		$\surd$		$\surd$
(1,(4,t),x)			$\surd$		$\surd$	$\surd$
(1,(4,f),x)			$\surd$		$\surd$	$\surd$
(1,(5,t),x)			$\surd$		$\surd$	$\surd$
(1,(5,f),x)			$\surd$		$\surd$	$\surd$
(1,6,x)		$\surd$		$\surd$		$\surd$
(1,7,x)		$\surd$		$\surd$		$\surd$
(3,8,a)	$\surd$	$\surd$		$\surd$	$\surd$	$\surd$
(7,8,a)	$\surd$	$\surd$		$\surd$	$\surd$	$\surd$
(6,6,x)	$\surd$	$\surd$		$\surd$		$\surd$
(6,7,x)		$\surd$		$\surd$		$\surd$
(6,(5,t),x)			$\surd$		$\surd$	$\surd$
(6,(5,f),x)			$\surd$		$\surd$	$\surd$
Satisfying Paths	$\{P_1, P_2\}$	$\{P_1, P_2\}$	$\{P_1, P_2\}$	$\{P_1, P_2\}$	$\{P_1, P_2\}$	$\{P_1, P_2\}$

Fig. 4. An Example.

. While it may be possible to find a minimal number of complete paths that satisfy a given test coverage criterion, this approach cannot always be used to minimize the testing effort. This is because some of the complete paths may be *infeasible.* For example, the complete path 1-2-4-8 for the program in Fig. 4 is infeasible.



Complete Paths:

$P_1 : 1-2-3-4-5-6-8-2-3-5-6-7-8-9$
$P_2 : 1-2-3-5-6-8-9$

Complete Path $\rightarrow$ / $du-paths$ for $(2,5,x)\downarrow$	P_1	P_2
2-3-4-5	$\surd$	
2-3-5	$\surd$	$\surd$

Complete Path $\rightarrow$	P_1	P_2
$du-association$ $(2,5,x)$ influences output y		$\surd$

Fig. 5. All-du-paths and Oi-all-uses Coverage Criteria.

Let us consider another example to illustrate the *all-du-paths* and *oi-all-uses* criteria. Fig. 5 shows two complete paths through the given control flow graph. The definition-use association $(2,5,x)$ is established along two $du-paths$ $2-3-4-5$ and $2-3-5$. The path P_1 exercises $(2,5,x)$ along both du-paths and therefore sufficiently tests it to meet the *all-du-paths* criterion. However, since the value of x at node 5 does not influence the output y along P_1, it does not meet the *oi-all-uses* criterion for $(2,5,x)$. On the other hand, although path P_2 exercises only one of the du-paths, the value of x at node 5 does influence the output y. Therefore while P_2 does not satisfy the *all-du-paths* criterion for $(2,5,x)$ it does satisfy the *oi-all-uses* criterion.

3. COMPLETE TESTING

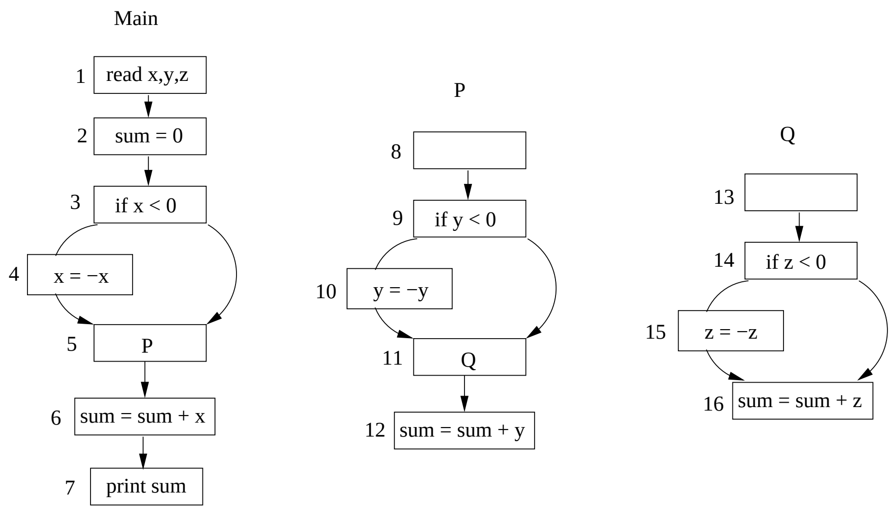
Before the software can be deployed it should be throughly tested. For large software, testing is carried out in a number of steps. First the individual procedures are tested one at a time during *unit testing.* The interfaces between the procedures are tested next during *integration testing.* A number of researchers have explored data flow testing methodologies for complete testing. Some of these techniques are suitable for unit testing as they are *intraprocedural* (i.e., their applicability is restricted to procedure bodies) [4; 15; 31]. Others have developed integration testing techniques that are *interprocedural* and therefore these techniques consider definition-use associations that extend across procedure boundaries [8; 26].

In this section we present an overview of unit and integration testing assuming that the *all-uses* criterion is being employed. During each step in testing we must identify the test requirements in form of definition-use associations that are to be tested. Once the test requirements have been identified, we must generate test cases to exercise the definition-use associations. One approach is to select a path along which at least one untested definition-use association is exercised. Test data is generated that causes the program to take the desired path and the definition-use association is tested. This process is repeatedly applied to test all of the test requirements.

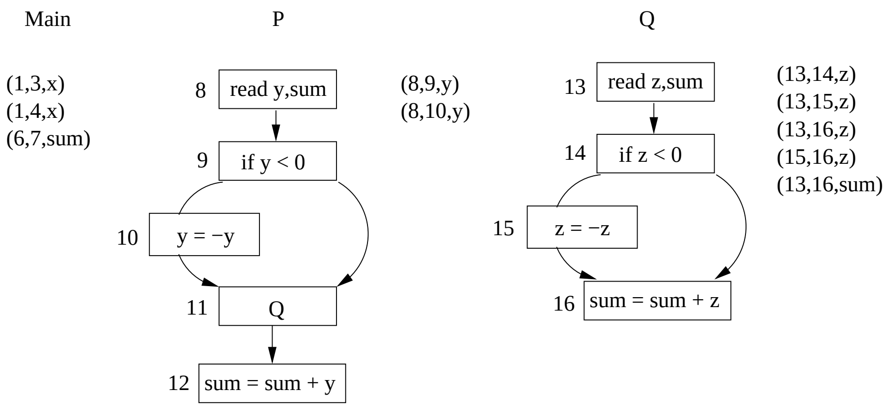
Unit testing considers one procedure at a time and exercises the definition-use associations within that procedure. A definition-use association within a procedure, also referred to as a *intraprocedural* definition-use association, is one for which the definition and the use belong to the procedure and a definition-clear path can be found within the procedure without examining the code belonging to any called procedures. In other words, to identify intraprocedural definition-use associations for a procedure, we conservatively assume that all variables visible to the called procedures are killed by them. A procedure may read values of global variables and may have input parameters. Definitions for input parameters and globals are introduced at the beginning of the procedure in form of read statements. These definitions also give rise to definition-use associations that must be tested. Once the definition-use associations have been computed the testing of the procedure can proceed using the path selection based strategy described earlier.

Integration testing exercises the interprocedural definition-use associations during one or more integration steps. An interprocedural definition-use association may be formed in a situation where the definition and use belong to different procedures. It can also be formed in cases where although the definition and use are in the same procedure, all of the definition-clear paths between them contain procedure calls. In each integration step one or more procedures are selected for integration. The program call graph is used to drive the selection process. For example, integration can be carried out in a top-down fashion or bottom-up fashion according to the call graph. In each integration step the interprocedural definition-use associations relevant for that step must be tested. The computation of interprocedural definition-use associations can be carried in a couple of different ways. One approach analyzes the complete program and computes all of the definition-use associations prior to integration testing [27]. Another approach partially analyzes the program to compute the relevant interprocedural definition-use associations during an integration step using demand-driven data flow analysis [10].

The example in Fig. 6 illustrates unit and integration testing. The sample program contains four variables which are all globals. The definition-use associations that are generated as requirements for unit testing are given in Fig. 6b. As we can see, in order to compute definition-use associations to unit test P and Q, definitions for globals referenced by these procedures are introduced in the entry nodes



(a) A Sample Program.



(b) Requirements for Unit Testing.

Integrate Q with P	Integrate P with Main
(8,16,sum)	(2,16,sum)
(16,12,sum)	(12,6,sum)
(10,12,y)	(1,9,y)
	(1,10,y)
	(1,14,z)
	(1,15,z)

(c) Requirements for Integration Testing Steps.

Fig. 6. An Example of Complete Testing.

of the procedures. During integration testing we assume that a bottom-up strategy is used. Therefore we first integrate Q into P and then we integrate P and $Main$. The interprocedural definition-use associations that must be tested during the integration steps are given in Fig. 6c.

In order to carry out the testing of the requirements during unit and integration testing we must generate test inputs. The test data generation problem is discussed later in this chapter. After the program is executed on a test input we would like to identify all the definition-use associations exercised by that input. For this purpose we can instrument the program to either generate an execution trace which can be later analyzed or we can dynamically track the definition-use associations that are being exercised. Techniques used by dynamic slicing algorithms can be used for this purpose [1; 30].

4. REGRESSION TESTING

Program changes are sometimes required after the deployment of software. There are two common reasons for such changes. First, although the software may have been completely tested prior to its deployment, some errors may only be discovered once the software is in use. Second, some changes may be made to better meet the needs of the customer. Regression testing is the process of testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested code. Therefore regression testing must test both the modified code and other parts of the program that may be affected by the program change. Researchers have developed a number of data flow based regression testing techniques [20; 33; 37].

In this section we describe an approach for regression testing proposed by Gupta *et al.* [20]. In this approach a *program change* is translated into a series of low level *program edits*. *Demand-driven static slicing* algorithms are used to identify definition-use associations that are *affected* by each program edit. The complete set of definition-use associations that are affected by a series of program edits corresponding to a program change are retested during regression testing. During regression testing we make use of existing test cases that were developed during the initial exhaustive testing of the software by maintaining the test suite [25]. However, in general, some new test cases may also have to be generated to adequately test the modified code.

The program change is mapped into a series of low level program edits. We identify a complete set of program edits such that changes to the program can always be mapped into a series of program edits. This list of edits is given below. As we can see, the edits are defined in terms of operations on the program control flow graph.

(1) Delete a use of a variable from an assignment statement.
(2) Insert a use of a variable in an assignment statement.
(3) Delete a definition of a variable from an assignment statement.
(4) Insert a definition of a variable in an assignment statement.
(5) Delete a use of a variable from a conditional statement.
(6) Insert a use of a variable in a conditional statement.
(7) Change an operator or a constant in an assignment statement.

(8) Change an operator or a constant in a conditional statement.
(9) Insert an edge.
(10) Delete an edge.
(11) Insert a new assignment statement.
(12) Insert a new conditional statement.

Let us assume that we used the *all-uses* criterion during complete testing. During *regression testing*, given a program edit, not only should the newly created definition-use associations be tested, but in addition all definition-use associations that are affected directly or indirectly by the program edit must be retested. The definition-use associations that should be tested during regression testing are classified into the following three categories:

. *New associations.* A program edit can cause the creation of new definition-use associations which must be tested. In particular, edit types 2, 3, 4, 6, 9, 11, and 12 can create new definition-use associations.

. *Value associations.* If an existing definition of a variable is directly or indirectly impacted by a program edit, such that the value assigned in the definition is changed, then all definition-use associations for this definition must be retested.

. *Path associations.* If a program edit directly or indirectly affects a predicate, then definition-use associations impacted by the predicate are also tested. A predicate is affected by a program edit if either the edit explicitly modifies the predicate or the predicate contains a use belonging to a *new* or *value* association. A definition-use association is impacted by a predicate if the definition in the association is control dependent upon the predicate.

The example in Fig. 7, taken from [20], illustrates the above approach. This program iteratively computes the square root of x. It has an error in statement 8 in which $x1$ should be assigned the value of $x3$ instead of $x2$. Correcting the error can be expressed in terms of two low level edits: deletion of the use of $x2$ from statement 8 followed by the insertion of the use of $x3$ in its place. The definition-use associations that must be tested due to these edits are as follows:

. *New associations.* The insertion of the new use of $x3$ in statement 8 creates a new definition-use association $(5, 8, x3)$.

. *Value associations.* Since the value of $x1$ defined by statement 8 is affected, the value associations $(8, (4, t), x1)$, $(8, (4, f), x1)$, $(8, 5, x1)$, $(8, (6, t), x1)$, and $(8, (6, f), x1)$ must be tested. The use of $x1$ in statement 5 affects the definition of $x3$ and therefore value associations $(5, 7, x3)$, $(5, (6, t), x3)$, $(5, (6, f), x3)$ must also be tested. Finally the use of $x3$ in statement 7 affects the definition of $x2$ and thus the value associations $(7, (4, t), x2)$, $(7, (4, f), x2)$, and $(7, 5, x2)$ must also be tested.

. *Path associations.* The above value associations indicate that the predicates 4 and 6 are affected by the edits. Therefore path associations created by definitions that are control dependent upon predicates 4 and 6 must be tested. The definitions that are control dependent upon these predicates include statements 5, 7, and 8. Since all definition-use associations for these three statements have already been marked for testing as value associations, all path associations are already included in the above set of value associations.

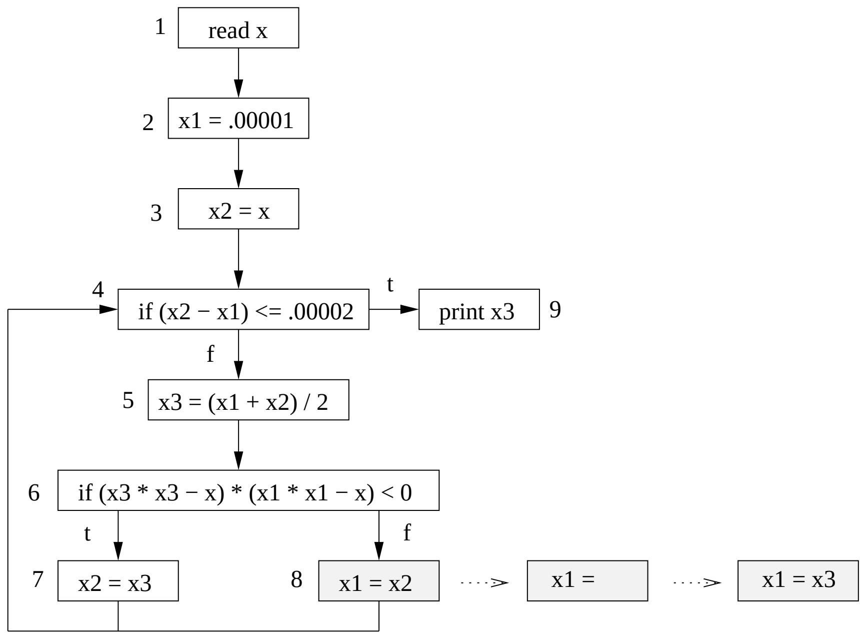


Fig. 7. Testing Following Program Edits.

The modified program must be analyzed to identify the definition-use associations that have to tested. One approach for performing such analysis is to exhaustively compute definition-use associations for the program prior to an edit as well as following an edit. By comparing the two the *new* definition-use associations are identified. By starting at the *new* associations and performing a forward static slicing operation over the definition-use associations, the *value* associations are identified. The predicates involved in *new* and *value* associations are now known and the definitions that are control dependent upon them are found. By performing forward static slicing operations from these definitions all the *path* associations are found.

Another approach that only partially computes the definition-use associations before and after the program edit is based upon demand driven analysis. Demand driven analysis algorithms can be devised to identify only those associations that are of interest. As an example consider a situation in which a definition of variable v has been deleted from node n. We would like to identify the *new* definition-use associations for v that may have been established. Using the algorithms shown in Fig. 8, we can compute the definitions of v that reach the entry of node n ($ReachingDefs(v,n)$) as well as the uses of v that are reachable from the exit of n (i.e., $ReachableUses(v,n)$). All definition-use associations for v that are established through n are now known. If some of these associations did not exist in the program prior to the edit, then they are *new* associations. The advantage of this approach is that the *new* associations have been found without exhaustively analyzing the program. The first algorithm for demand driven computation of definition-use

```
ReachingDefs (v, n) {
    Initialize Worklist to contain node n
    While Worklist is not empty do
          remove a node, say m, from Worklist
          if m defines variable v then
             add m to DefsFound
          else
             add predecessor nodes of m to Worklist
          endif
    endwhile
    return(DefsFound)
}
```

(a) Finding Reaching Definitions Using Backward Analysis.

```
ReachableUses (v, n) {
    Initialize Worklist to contain node n
    While Worklist is not empty do
          remove a node, say m, from Worklist
          if m contains a use of variable v then
             add m to UsesFound
          endif
          if m does not define variable v then
             add successor nodes of m to Worklist
          endif
    endwhile
    return(UsesFound)
}
```

(b) Finding Reachable Uses Using Forward Analysis.

Fig. 8. Reaching Definitions and Reachable Uses Analysis.

associations that we are aware of was proposed by Weiser who used this approach to compute static slices [39; 18]. Since then this approach has been used to deal with program edits in other works [18; 20].

5. TEST INPUT DATA GENERATION

One of the most difficult tasks one faces during the initial testing of a program is that of generating test cases to exercise the various test requirements. While during functional testing the test cases are generated from the functional specification, during data flow testing test cases must be generated to provide adequate coverage under the selected test criterion for a given implementation. One by one as the test cases are generated, the program is executed on them and the execution is analyzed to identify the test requirements that are covered by the test case. As testing proceeds, an attempt must be made to generate test cases that exercise test requirements that are yet to be covered.

5.1 Manual Test Input Generation

One approach to test input generation leaves the task of generating test cases to the person testing the software. Even under this situation some of the other tasks can still be carried out automatically. In particular, the determination of overall test requirements, the tracking of test requirements exercised by a given test case, and identification of test requirements that must be exercised to satisfy the test coverage criterion can all be carried out automatically.

The generation of test cases is extremely labor intensive and time consuming. Therefore the use of static analysis information has been explored to guide the user during the generation of test data [21]. In particular, static information is used to identify paths that cover one or more test requirements that are yet to be tested. Another problem that is encountered is due to the presence of infeasible program paths. Since the identification of test requirements is based upon static analysis, it is possible that some of the identified test requirements are in fact *infeasible*, that is, there is no program input on which they are exercised. The generation of infeasible test requirements is unavoidable because static analysis is based upon the assumption that all program paths are feasible. The time spent in trying to generate a test case for an infeasible test requirement is clearly wasted. In fact it is quite possible that the test engineer discovers that a test requirement is infeasible only after spending a great deal of time trying to generate a test case for the requirement.

To reduce the time wasted on infeasible requirements, an approach was proposed by Bodik *et al.* [2]. A simple static analysis is described to identify some of the infeasible test requirements. These requirements can be removed from consideration during the testing process. The basic idea behind this approach is to identify smallest subpaths through the control flow graph that are infeasible. The traditional data flow analysis for identifying definition-use associations is then modified such that it prevents propagation of data flow information along infeasible subpaths. Consider the example in Fig. 9 which contains an infeasible definition-c-use association $(1, 4, a)$. This association can be identified as being infeasible by the analysis in [2]. In particular, it is determined that if $x < y$ is true then $x < y + 1$ must also be true and therefore the subpath $1 - 2 - 4$ is infeasible.

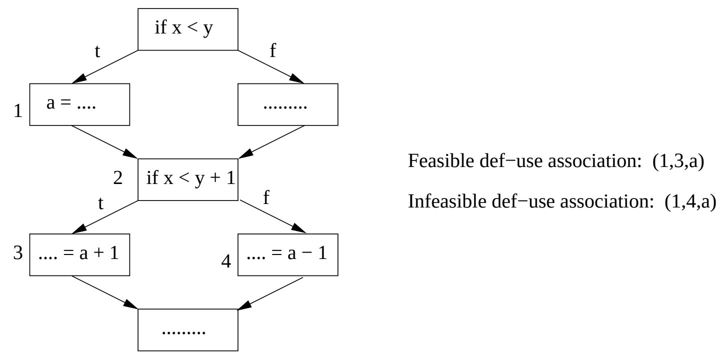


Fig. 9. Feasible vs Infeasible Definition-Use Associations.

5.2 Automated Test Input Generation

Since test input generation is a time consuming process, researchers have proposed techniques for automating test input generation [7; 12; 16; 29; 22; 23; 24]. In this section we describe an approach that has been proposed to automatically generate test data. We first consider a class of programs in which the branch predicates involve computations that can be expressed as linear functions of the program inputs. Later we show how this condition can be relaxed to allow for more general programs.

The high level view of the automated test data generation method that we present is as follows. We first select a program path that exercises the test requirement being considered. The task of the test generator is to identify a program input which will cause the program's execution to follow the selected path. To carry out this task we consider each branch predicate evaluation along the selected program path and by analyzing the code along the program path we derive a linear constraint on the program's inputs that must be satisfied for that branch predicate to evaluate correctly. This process is repeated for each branch predicate evaluation and a system of linear constraints is obtained. Any constraint solver, such as XA [43] or UNA [23], can then we used to solve the linear constraint system.

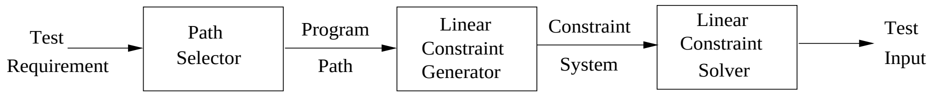


Fig. 10. Automated Test Data Generation and Testing.

The above approach deals with infeasible path problem quite naturally. If the path chosen is infeasible, then the set of constraints that result will be inconsistent and therefore no solution will be found. It is possible that there are certain constraints on the input domain (e.g., an input value may have to be a positive integer to be meaningful). These constraints can also be naturally handled in conjunction with the derived constraints by the constraint solver.

The key step of the above process is the derivation of the system of linear constraints from the program. We discuss two approaches to carry out this task: one is based upon *symbolic evlauation* [3] and the other is an *execution based* [24] approach. We illustrate these approaches through an example program of Fig. 11 which reads three input values into variables a, b and c and then changes the contents of the variables such that eventually a contains the smallest number and c contains the largest number. We consider the generation of test data for a couple of paths through this program, one feasible and another infeasible.

Symbolic evaluation based derivation of linear constraints. Let us consider the *symbolic evaluation* approach. For each branch predicate evaluation along the selected path, we prepare a straightline code segment consisting of all statements along the path except for the earlier branch predicate evaluations. This straightline code is symbolically executed so that the branch predicate evaluation under consideration can be expressed directly in terms of the program inputs. The desired

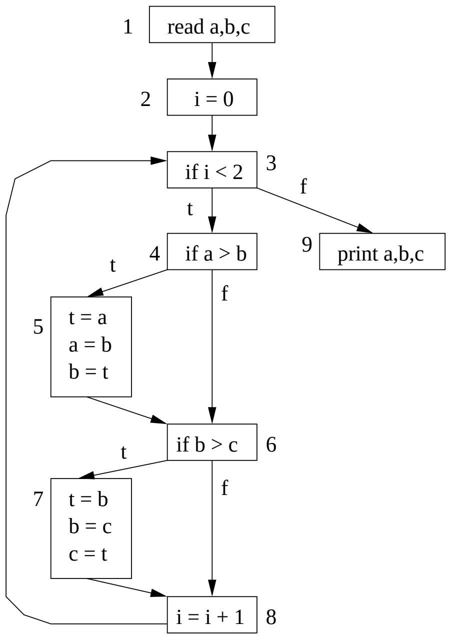


Fig. 11. An Example for Test Data Generation.

evaluation of the branch predicate imposes a constraint on the inputs that must be satisfied by the test input. This process is repeated for each branch predicate evaluation along the selected path and a system of constraints is derived.

Fig. 12 illustrates this approach for the sorting example of Fig. 11. The first path considered is a feasible path. The code along the path is given and the desired branch predicate evaluations are indicated. For simplicity the statements that implement the loop, causing it to iterate twice, are ignored. This is because no input constraints will result from analyzing the loop predicate evaluations since the number of times the loop iterates is fixed and independent of the program input. This fact can also be easily discovered through symbolic evaluation. When symbolic evaluation of the code along the path to other branch predicate evaluations is carried out we obtain a system of constraints shown in the figure. This indicates that for the path to be followed, we must select input values for a, b and c such that $a > b > c$.

If we consider the second path shown in the figure, we notice that this path only differs from the first path in its evaluation of the last branch predicate (true instead of false). Therefore while the first three constraints are the same, the fourth constraint we obtain is different ($b > a$ instead of $b \not> a$). We can clearly see that the constraint system that we have obtained now has no solution since the constraints corresponding to the first and last branch predicate evaluations cannot be simultaneously satisfied. Therefore we can conclude that the selected path is infeasible.

Test data for feasible path: 1-2-3(t)-4(t)-5-6(t)-7-8-3(t)-4(t)-5-6(f)-8-3(f)-9

read a,b,c	read a,b,c	read a,b,c	read a,b,c	read a,b,c
if a > [true]	if a > b [true]	t = a	t = a	t = a
t = a	$\Downarrow$	a = b	a = b	a = b
a = b	$\Downarrow$	b = t	b = t	b = t
b = t	$\Downarrow$	if b > c [true]	t = b	t = b
if b > c [true]	$\Downarrow$	$\Downarrow$	b = c	b = c
t = b	$\Downarrow$	$\Downarrow$	c = t	c = t
b = c	$\Downarrow$	$\Downarrow$	if a > b [true]	t = a
c = t	$\Downarrow$	$\Downarrow$	$\Downarrow$	a = b
if a > b [true]	$\Downarrow$	$\Downarrow$	$\Downarrow$	b = t
t = a	$\Downarrow$	$\Downarrow$	$\Downarrow$	if b > c [false]
a = b	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
b = t	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
if b > c [false]	a > b	a > c	b > c	b $\not>$ a
print a,b,c				

$$\begin{array}{l} a > b \\ a > c \\ b > c \\ b \not> a \end{array} \quad \Rightarrow \quad a > b > c$$

Test data for infeasible path: 1-2-3(t)-4(t)-5-6(t)-7-8-3(t)-4(t)-5-6(t)-7-8-3(f)-9

read a,b,c	read a,b,c	read a,b,c	read a,b,c	read a,b,c
if a > [true]	if a > b [true]	t = a	t = a	t = a
t = a	$\Downarrow$	a = b	a = b	a = b
a = b	$\Downarrow$	b = t	b = t	b = t
b = t	$\Downarrow$	if b > c [true]	t = b	t = b
if b > c [true]	$\Downarrow$	$\Downarrow$	b = c	b = c
t = b	$\Downarrow$	$\Downarrow$	c = t	c = t
b = c	$\Downarrow$	$\Downarrow$	if a > b [true]	t = a
c = t	$\Downarrow$	$\Downarrow$	$\Downarrow$	a = b
if a > b [true]	$\Downarrow$	$\Downarrow$	$\Downarrow$	b = t
t = a	$\Downarrow$	$\Downarrow$	$\Downarrow$	if b > c [true]
a = b	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
b = t	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
if b > c [true]	a > b	a > c	b > c	b > a
print a,b,c				

$$\begin{array}{l} a > b \\ a > c \\ b > c \\ b > a \end{array} \quad \Rightarrow \quad \text{Infeasible Path}$$

Fig. 12. Constraint Derivation Using Symbolic Evaluation.

Execution based derivation of linear constraints. Next we consider an *execution based* approach for deriving the constraints. Given a boolean predicate, we can write this predicate in a normalized form P *relop* 0, where *relop* is a relational operator. Our goal is to find a linear representation for P in terms of the program inputs. If P is a linear function of program inputs, it can be expressed in the following form where i_1 through i_n are the inputs and x_0 through x_n are constants:

$$P = x_0 + x_1 \times i_1 + x_2 \times i_2 + \cdots + x_n \times i_n.$$

If we can find the values of the constants in the above representation we will have the linear representation of P in terms of the program inputs. In order to do so we execute the statements along the path leading to P, excluding the prior branch predicates, on $n+1$ arbitrary input data sets and print out the values of P for each of these input data sets. By substituting the input data values and the corresponding printed value of P in the above equation we obtain the following system of equations:

$$P[0] = x_0 + x_1 \times i_1[0] + x_2 \times i_2[0] + \cdots + x_n \times i_n[0]$$

$$P[1] = x_0 + x_1 \times i_1[1] + x_2 \times i_2[1] + \cdots + x_n \times i_n[1]$$

$$\ldots$$

$$P[n] = x_0 + x_1 \times i_1[1] + x_2 \times i_2[n] + \cdots + x_n \times i_n[n].$$

Therefore we now have a system of $n+1$ equations in $n+1$ unknowns which can be solved to compute the values of x_0 through x_n using standard packages such as MATLAB [42]. The result of the above computation is the linear representation of P in terms of program inputs.

Fig. 13 illustrates the derivation of a predicate expression using the execution based approach. In this example we derive the linear representation of the last predicate expression corresponding to the feasible path of Fig. 12. The predicate is represented by $f(a,b,c)$ in terms of the three program inputs. The code along the path is executed on some arbitrarily selected inputs and from the resulting values of f, we obtain the system of equations from which the values of constant coefficients are computed. As we can see, this approach yields the constraint $b \not> a$ which is same as the earlier representation that we obtained using the symbolic execution approach.

There are some advantages to using the execution based approach. In the example we considered, it was fairly straightforward to carry out the symbolic evaluation because variable names being referenced were statically unambiguous, there were no array or pointer references in the code, and finally the expressions being manipulated were small and simple. For long paths containing array and pointer references symbolic evaluation can become impossible to carry out. However, the execution based approach is free of these limitations.

Non-linear Constraints. We have restricted our discussion to situations in which the predicate expressions were linear functions of the program inputs. In general this may not be the case and some of the predicate expressions may be non-linear functions of the inputs. The presence of non-linear predicate expressions poses challenges both in deriving the predicate expressions and solving the non-linear

```
read a,b,c
t = a
a = b
b = t
t = b
b = c
c = t
t = a
a = b
b = t
if b > c [false]
```

if b > c [false]

$\Downarrow$

if b − c > 0 [false]

$\Downarrow$

if f(a,b,c) > 0 [false]

where, $f(a,b,c) = x_1 \times a + x_2 \times b + x_3 \times c$

$\Downarrow$

```
f(a,b,c) {
   read a,b,c
   t = a
   a = b
   b = t
   t = b
   b = c
   c = t
   t = a
   a = b
   b = t
   print (f(a,b,c) = b - c)
}
```

$f(0,5,10) = 5$
$f(1,6,11) = 5$
$f(2,8,10) = 6$
$f(3,9,12) = 6$

$$
\begin{aligned}
f(0,5,10) = 5 &\Rightarrow x_0 + x_1 \times 0 + x_2 \times 5 + x_3 \times 10 = 5\\
f(1,6,11) = 5 &\Rightarrow x_0 + x_1 \times 1 + x_2 \times 6 + x_3 \times 11 = 5\\
f(2,8,10) = 6 &\Rightarrow x_0 + x_1 \times 2 + x_2 \times 8 + x_3 \times 10 = 6\\
f(3,9,12) = 6 &\Rightarrow x_0 + x_1 \times 3 + x_2 \times 9 + x_3 \times 12 = 6
\end{aligned}
$$

$$
\begin{array}{lcl}
x_0 + 5 \times x_2 + 10 \times x_3 = 5 & & x_0 = 0\\
x_0 + x_1 + 6 \times x_2 + 11 \times x_3 = 5 & \Rightarrow & x_1 = -1\\
x_0 + 2 \times x_1 + 8 \times x_2 + 10 \times x_3 = 6 & & x_2 = 1\\
x_0 + 3 \times x_1 + 9 \times x_2 + 12 \times x_3 = 6 & & x_3 = 0
\end{array}
$$

$f(a,b,c) = x_0 + x_1 \times a + x_2 \times b + x_3 \times c = b - a \quad \Rightarrow \quad$ if b > a [false] $\quad \Rightarrow \quad$ b $\not>$ a

Fig. 13. Execution Based Constraint Derivation.

constraint system. In principle, the symbolic evaluation approach is not restricted to a linear representation. However, the simplification of non-linear expressions during symbolic evaluation is far more complex. The execution based approach has a rather strict limitation. It requires that the form of the predicate expression in terms of program inputs be known. However, it is not realistic to assume that this would be the case. Even if the non-linear predicate expressions are identified, solving the resulting system of non-linear constraints is a problem.

An approach that addresses the above problem was proposed by Gupta *et al.* in [24; 23]. The predicate expressions are handled as follows.

. All predicate expressions are treated uniformly by assuming that they are linear functions of the inputs. The execution based approach is used to derive these linear representations. If the predicate expression is linear, we have chosen the correct form. However, if a predicate expression has a non-linear representation, the linear representation that is derived is an approximation of the actual non-linear representation.

. A non-linear function cannot be accurately approximated by a linear function across the entire input domain. This problem is addressed as follows. Rather than having a single linear representation for the predicate expression for all program inputs, the linear representation is derived with respect to each program input. In fact the linear approximation represents the tangent plane to the non-linear expression for a given input. Since the approximate linear representation is specific to a program input, a specific input must be chosen before the representations can be derived.

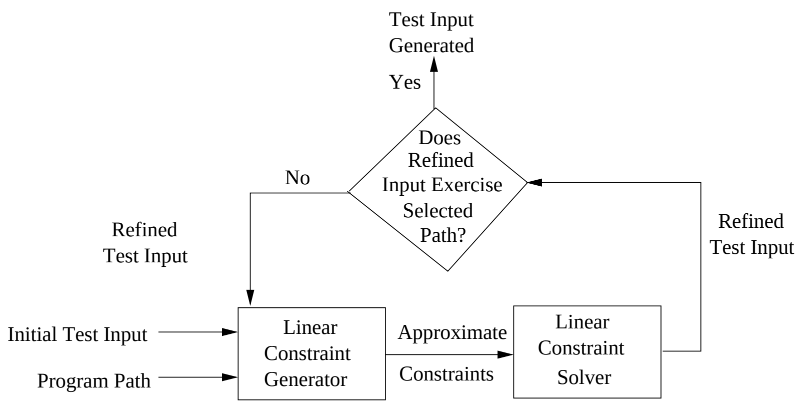


Fig. 14. Iterative Refinement of Test Data.

The above treatment of predicate expressions leads to the overall approach shown in Fig. 14 that uses an iterative relaxation method to generate test input. Test data generation is initiated with an arbitrarily chosen input from a given domain. This input is then iteratively refined to obtain an input on which all the predicate expressions on the given path evaluate to the desired outcome. In each iteration the program statements relevant to the evaluation of each branch predicates on the path are executed, and a set of linear constraints is derived. The constraints are then solved to obtain the increments for the input. These increments are added to the current input to obtain the input for the next iteration. The relaxation technique used in deriving the constraints provides feedback on the amount by which each input variable should be adjusted for the branches on the path to evaluate to the desired outcome.

6. CONCLUDING REMARKS

In this chapter we provided an overview of different aspects of data flow based testing of programs. We described the various data flow based test coverage criterion and the relationships amongst them. As we show the various aspects of data flow testing can be automated to a significant extent. In particular, static data flow analysis is used to identify test requirements, dynamic analysis is used to track test requirements exercised during a given program execution, the test requirements that remain to be tested can be automatically tracked, and finally it can be automatically determined when enough testing has been performed to satisfy a given test coverage criterion. We also described techniques that automate the difficult task of generating test cases. While we considered the testing of sequential programs, researchers are also developing testing strategies for dealing with object-oriented programs [28], parallel and real-time programs [38; 5; 11].

References

[1] H. Agrawal and J.R. Horgan, "Dynamic Program Slicing," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] R. Bodik, R. Gupta, and M.L. Soffa, "Refining Data Flow Information using Infeasible Paths," *ACM SIGSOFT 5th Symposium on Foundations of Software Engineering and 6th European Software Engineering Conference*, LNCS 1301, Springer Verlag, pages 361–377, September 1997.

[3] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 215–222, September 1976.

[4] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, pages 1318–1332, November 1989.

[5] J.C. Corbett, "Timing Analysis of Ada Tesking Programs," *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996.

[6] R. De Millo, W. McCracken, R. Martin, and J. Passafiume, *Software Testing and Evaluation*, Benjamin/Cummings, Menlo Park, CA, 1987.

[7] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pages 900–910, September 1991.

[8] E. Duesterwald, R. Gupta, and M.L. Soffa, "A Demand-Driven Analyzer for Data Flow Testing at the Integration Level," *SIGSOFT/IEEE 18th International Conference on Software Engineering*, pages 575–584, 1996.

[9] E. Duesterwald, R. Gupta, and M.L. Soffa, "Rigorous Data Flow Testing through Output Influences," *2nd Irvine Software Symposium*, pages 131–145, March 1992.

[10] E. Duesterwald, R. Gupta, and M.L. Soffa, "A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6, pages 992–1030, November 1997.

[11] M.B. Dwyer and L.A. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs," *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 62–75, December 1994.

[12] R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation," *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 1, pages 63–86, January 1996.

[13] P.G. Frankl and E.J. Weyuker, "Provable Improvements on Branch Testing," *IEEE Transactions on Software Engineering*, Vol. 19, No. 10, pages 962–975, October 1993.

[14] P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pages 1483–1498, October 1988.

[15] P.G. Frankl and E.J. Weyuker, "An Analytical Comparison of the Fault-detecting Ability of Data Flow Testing Techniques," *SIGSOFT/IEEE 15th International Conference on Software Engineering*, pages 415–424, 1993.

[16] M.J. Gallagher and V.L. Narsimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, pages 473–484, August 1997.

[17] T.L. Graves, M.J. Harrold, J-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *ACM Transactions on Software Engineering and Methodology*, Vol. 10, No. 2, pages 184–208, April 2001.

[18] R. Gupta and M.L. Soffa, "A Framework for Partial Data Flow Analysis," *IEEE International Conference on Software Maintenance*, pages 4–13, September 1994.

[19] R. Gupta and M.L. Soffa, "Priority Based Data Flow Testing," *IEEE International Conference on Software Maintenance*, pages 348–357, October 1995.

[20] R. Gupta, M.J. Harrold, and M.L. Soffa, "An Approach to Regression Testing using Slicing," *IEEE International Conference on Software Maintenance*, pages 299–308, November 1992.

[21] R. Gupta and M.L. Soffa, "Employing Static Information in the Generation of Test Cases," *Journal of Software Testing, Verification and Reliability*, Vol. 3, pages 29–48, 1993.

[22] N. Gupta, A.P. Mathur, and M.L. Soffa, "Generating Test Data for Branch Coverage", *15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

[23] N. Gupta, A.P. Mathur, and M.L. Soffa, "UNA Based Iterative Test Data Generation and its Evaluation", *14th IEEE International Conference on Automated Software Engineering*, pages 224–232, October 1999.

[24] N. Gupta, A.P. Mathur, and M.L. Soffa, "Automated Test Data Generation Using an Iterative Relaxation Method", *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering*, pages 231–244, November 1998.

[25] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of the Test Suite," *ACM Transactions on Software Engineering and Methodology*, Vol. 2, No. 3, pages 270–285, July 1993.

[26] M.J. Harrold and M.L. Soffa, "Interprocedural Data Flow Testing," *ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 158–167, 1989.

[27] M.J. Harrold and M.L. Soffa, "Efficient Computation of Interprocedural Definition-Use Chains," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 2, pages 175–204, March 1994.

[28] P.C. Jorgensen and C. Erickson, "Object-Oriented Integration Testing," *Communications of the ACM*, Vol. 37, No. 9, pages 30–38, September 1994.

[29] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pages 870–879, August 1990.

[30] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, Vol. 29, pages 155–163, October 1988.

[31] J.W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, pages 347–354, May 1983.

[32] T.J. McCabe, "Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," *National Bureau of Standards* (now NIST), Publication 500-99, Washington, 1982.

[33] T. Ostrand and E.J. Weyuker, "Using Data Flow Analysis for Regression Testing," *Sixth Annual Pacific Northwest Software Quality Conference*, pages 58–71, September 1988.

[34] A.S. Parrish and S.H. Zweben, "On the Relationships Among the All-uses, All-DU-paths, and All-edges Testing Criteria," *IEEE Transactions on Software Engineering*, Vol. 21. No. 12, pages 1006–1009, December 1995.

[35] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pages 367–375, April 1985.

[36] G. Rothermel, R.H. Untch, C-C. Chu, and M.J. Harrold, "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, pages 929–948, October 2001.

[37] A.M. Taha, S.M. Thebut, and S.S. Liu, "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis," *COMPSAC*, pages 527–534, September 1989.

[38] R.N. Taylor, D.L. Levine, and C.D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, March 1992.

[39] M.D. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352–357, April 1988.

[40] E.J. Weyuker, "More Experience with Data Flow Testing," *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, pages 912–919, September 1993.

[41] E.J. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, pages 121–128, Feb. 1990.

[42] "MATLAB," *The MathWorks*, http://www.mathworks.com.

[43] "XA," *Sunset Software Technology*, http://www.sunsetsoft.com.