Priority Based Data Flow Testing*

Rajiv Gupta and Mary Lou Soffa
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{gupta,soffa}@cs.pitt.edu

Abstract

Software testing is an expensive component of software development and maintenance. For data flow testing, test cases must be found to test the def-use pairs in a program. Since some of the def-use pairs identified through static analysis may be infeasible, no amount of testing effort may result in exhaustive testing of a program. Therefore in practice a fixed amount of effort is spent in testing a program. In this paper we develop an approach for assigning priorities to def-use pairs, such that the def-use pairs with higher priorities can be expected to require less effort for test case generation and therefore testing. Thus, by using the priorities as a guide for ordering the def-use pairs for testing, we can maximize the number of def-use pairs tested using a fixed amount of testing effort. We apply the technique to regression testing during the software maintenance phase, in which case the priorities are assigned to capture not only the difficulty in test case generation but also the likelihood that an error introduced by a program change will be uncovered by the test case.

Keywords - *software testing, data flow analysis, def-use testing, regression testing, control dependences, definition-free paths.*

1 Introduction

Structural testing is widely recognized as being an effective technique to test programs, and includes path and data flow testing techniques. In structural testing, various criteria are used to guide the testing of programs. Using a particular criteria, certain requirements are identified as points in the programs that need to be covered by a test case. For data flow testing, these requirements are a set of definition-use pairs. Typically the program is not exhaustively tested under a given criteria but rather a subset of the criteria is tested. There are two reasons for the lack of complete coverage. One reason is that some of the requirements may be infeasible (for example, certain paths in the program may be infeasible and the def-use pairs established only along infeasible paths are also infeasible). Another reason is that testing is expensive in terms of user effort in finding test cases, and thus is typically allotted only a certain amount of time. In developing the test cases the user would benefit from some guidance in ordering the requirements that need to be covered as well as showing the data flow paths that must be traversed to cover a requirement. It could be that the user chooses a particularly difficult requirement and tries to find a test case to cover the requirement using a large portion of the time allocated for this one requirement. Therefore techniques that maximize the effectiveness of testing by guiding the user on how to spend the testing effort would be helpful.

The goal of this work is to develop heuristics for prioritizing testing requirements so that the effectiveness of testing is maximized. In this paper, we present a priority based testing technique that enables a priority to be placed on each of the requirements for data flow testing. The priorities developed guide the users testing the program in their search for test cases that satisfy the requirements. The priority of a def-use pair is defined to capture the difficulty that we can expect to encounter in generating a test case for the def-use pair. The higher the priority, the lower is the expected difficulty in generating the test case. There are two factors that contribute to the priority of a def-use pair. The first factor is the difficulty in finding a path from the start of the program that first passes through the definition and then through the use. This factor is dependent upon the number of predicates along the path whose appropriate outcome is required to ensure that the path is followed. The second factor constitutes the difficulty of finding a path from the definition to the use along which the def-use pair is established (i.e., definition free paths). This factor is computed by comparing the number of total data flow paths from a definition to a use with the number of these paths along which the def-use pair is established. The primary idea is that a greater number of def-use paths existing between the two nodes results in a better chance of

*Supported in part by the National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the Univ. of Pittsburgh.

finding a test case that goes though the two nodes.

Our technique is not only useful for the initial testing of a program but also during regression testing following program changes. During regression testing, new test cases may be required to test newly created def-use pairs or existing def-use pairs that have been effected by a program change. Therefore there is a need for prioritizing the def-use pairs according to the difficulty of generating test cases. A goal of regression testing is to ensure that program changes have not introduced new errors in the program. Therefore the priorities must also depend upon the likelihood that the testing of a def-use pair will uncover any newly introduced errors. The approach suggested in this paper computes priorities such that preference is given to def-use pairs for which the generation of test cases is likely to be easier and the execution of the test case is likely to uncover new errors.

The priority based testing technique uses a control flow graph representation of a program where each node represents a single program statement. We first establish definitions for control predicates, data flow paths, and def-use paths and develop algorithms for computing these paths for acyclic graphs. Using these characteristics, a priority is computed for each of the requirements that was determined by the def-use based testing criteria. These priorities guide the order in which test cases are sought by the user testing the program. In order to handle cyclic graphs we first apply loop unwinding transformations to create acyclic graphs in which all paths of interest are established and then we use our algorithms for acyclic graphs for computing data flow and def-use paths.

It has been estimated that about 50% of software development costs can be attributed to testing [2, 9]. The cost of testing is due to the data flow analysis required to identify requirements, generation of test cases, and the execution of test cases. In order to reduce the cost of computing and maintaining data flow information incremental as well as slicing based approaches have been suggested [6, 5]. In order to minimize the cost of test case execution during maintenance, regression testing techniques have been developed [10, 6]. It is well accepted that test case generation is one of the most time consuming aspect of software testing [11]. Heuristics have been developed that use static information to guide the test generation process for a given requirement [4]. Also static analysis techniques that group requirements that can be potentially satisfied by the same test case have been considered [1, 3]. The approach of prioritizing the requirements can be used in conjunction with the above techniques to maximize the testing achieved using a fixed amount of effort. However, to our knowledge, no work has been done on prioritizing the testing requirements under a given criterion.

In section 2 we present the definitions, the graph transformation technique and the technique for the initial testing of the program during development. Section 3 demonstrates how the technique can be used during regression testing, when only the changes in a program are to be tested. Finally, concluding remarks are given in section 4.

2 Priority-based Testing

In this section we present priority based data flow testing used for the initial testing of a program. We begin by focusing on definitions for the number of data flow paths and the number of definition-use paths that occur from a node containing a definition to a node containing a use. We also define the number of control predicates that must evaluate a certain way for an execution path to be established which visits the definition and then the use. These numbers are used in defining the testing priorities.

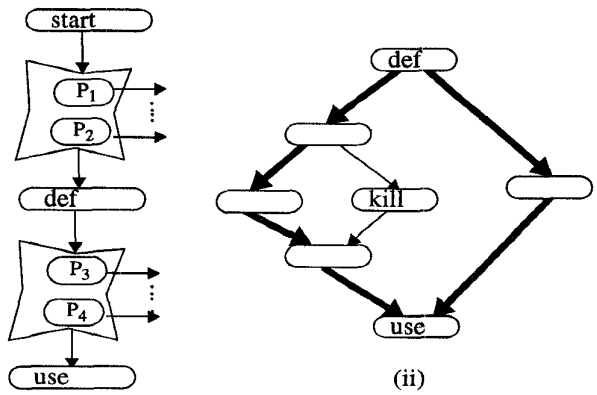


Figure 1: Defining Priority of a def-use pair: (i) The Role of Control Predicates; (ii) The Role of Definition-Free Paths.

Given a definition use pair, we intuitively define the priority of the pair to be the difficulty of establishing a path along which a def-use pair is created. In order for the program's execution to take a desired path certain predicates must evaluate appropriately. These predicates are among those predicates in the program upon which the definition and/or the use statement are directly or indirectly control dependent. In Figure 1(i) we show how inappropriate evaluation of certain predicates may cause the execution not to exercise the def-use pair of interest. Each predicate imposes an additional constraint on the inputs that must be satisfied for the program to follow an execution path that visits the definition and then the use. The greater the number of such control predicates, the tougher it is to establish a path that visits both the definition and the use and therefore the lower is the priority of the def-use pair. Even if the above predicates evaluate appropriately and execution passes through definition

and use statements, additional conditions may have to be met to ensure that the def-use pair is exercised. In Figure 1(ii) we show a situation in which there are three paths from the definition to the use. However, only along two of the paths is the def-use pair actually established. The greater the number of paths along which the def-use pair is exercised in relation to the total number of paths from def to use, the easier it is to find such a path and therefore the priority of the def-use pair is higher.

Next we provide precise definitions for the above concepts and then define the priorities of def-use pairs. We first consider acyclic graphs and later extend our analysis to allow loops in programs.

Definition 1: The number of *relevant control predicates* from node *src* to node *dest*, denoted by $ControlPreds(src, dest)$, is the maximum number of control predicates visited along a path from *src* to *dest* such that node *dest* is directly or indirectly control dependent upon these predicates.

Definition 2: The number of *data flow paths* from node *src* to node *dest*, denoted by $DataFlowPaths(src, dest)$, is the number of paths from *src* to *dest* that represent unique and distinctive paths in terms of data flow.

Definition 3: The number of *definition-use paths* from a definition of variable v at node *def* to the use of variable v at node *use*, denoted by $DefUsePaths(def, use)$, is the number of distinctive paths in terms of data flow from *def* to *use* which are free of definitions of v.

Definition 4: The *priority* of a def-use pair (def, use), denoted by $Priority(def, use)$, is defined as follows:

$$Priority(def, use) = f_1(ControlPreds(start, def), ControlPreds(def, use)) \times f_2(DefUsePaths(def, use), DataFlowPaths(def, use)).$$

As we can see from the preceding definition of priority, the first factor accounts for the control predicates that influence establishing of a path that visits the definition and then the use. The second factor accounts for the definition-free paths between the definition and the use. For the purpose of discussion in the remainder of the paper, we assume the following choice of functions:

$$f_1 = \frac{1}{1+ControlPreds(start,def)+ControlPreds(def,use)}$$

and

$$f_2 = \frac{DefUsePaths(def,use)}{DataFlowPaths(def,use)}.$$

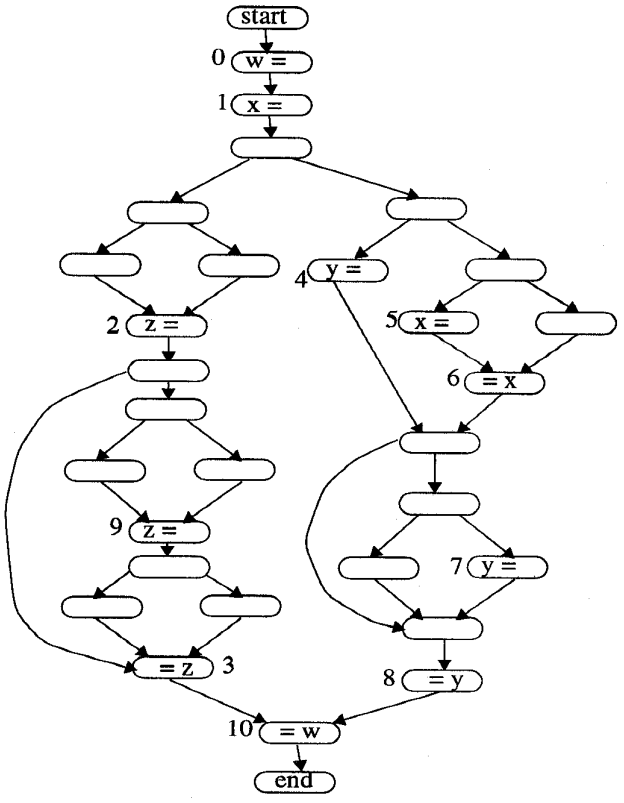
Given the above functions, if a definition use pair is exercised under all executions of a program, then the priority will evaluate to one. This is because in this situation

$ControlPreds(start, def) = 0$,
$ControlPreds(def, use) = 0$, and
$DefUsePaths(def, use) = DataFlowPaths(def, use)$.

In all other circumstances the priority is less than one.

An example illustrating the computation of priorities using the above specifications of f_1 and f_2 is shown in Figure 2. The flow graph contains seven def-use pairs whose priorities are shown. Consider the def-use pair for variable w. Since this def-use pair is exercised in all executions, its priority is one. The priorities for other def-use pairs are less than one. Consider the def-use pair for variable z from statement 2 to statement 3. The evaluation of a single predicate controls whether or not the execution will reach 2. Once statement 2 is reached we are guaranteed to reach 3. There are five paths from 2 to 3. However, the def-use pair is established only along one of the paths. Therefore the priority of this def-use pair is given by $\frac{1}{2} \times \frac{1}{5} = 0.1$. The priorities of other def-use pairs are similarly computed. It is interesting to note that the priorities of the def-use pairs vary quite significantly, the highest being 1 and the lowest being 0.1.

The data flow equations for computing the characteristics used for estimating priorities for acyclic graphs are given in Figure 3. The number of data flow paths from *src* to *dest*, $DataFlowPaths(src, dest)$, is the sum of the number of data flow paths from *src* to each of the immediate predecessors of *dest*. The number of def-use paths, $DefUsePaths(def_v, use_v)$, from a definition of variable v, def_v, to a use of variable v, use_v, is the sum of the number of definition free paths from def_v to each of the immediate predecessors of use_v that does not kill def_v. The computation of the maximum number of relevant control predicates along a path from *src* to *dest*, $ControlPredicates(src, dest)$, is computed as follows. The subgraph that includes *src*, *dest* and all nodes and edges that are a part of some path from *src* to *dest* are considered for this analysis. The predicates in the subgraph on which *dest* is directly or indirectly control dependent are identified. Using a single topological traversal over the subgraph the value of $ControlPredicates(src, dest)$ is then computed. Since the identification of control predicates requires control dependence information, we finally provide data flow equations for determining all predicate statements on which a node (n) is directly or indirectly control dependent $(AllControlAnc(n))$. First the immediate control ancestors $(ImmControlAnc)$ are identified and then closure over these sets is taken to identify $AllControlAnc$ sets. The computation of $ImmControlAnc$ requires the computation of post-dominator information $(PostDom)$. The algorithms for solving the above equations and computing priorities for acyclic graphs are given in Figures 4 and 5.



def-use pair	priority
w (0,10)	1.00
z (9,3)	0.33
x (5,6)	0.25
y (7,8)	0.25
y (4,8)	0.23
x (1,6)	0.17
z (2,3)	0.10

du- pair: w (0,10)
ControlPreds(start ,0) = 0
ControlPreds(0,10) = 0
DataFlow Paths (0,10) = 19
Def Use Paths (0,10) = 19
Priority (0,10) = 1.00

du- pair: x (1,6)
ControlPreds(start ,1) = 0
ControlPreds(1,6) = 2
DataFlow Paths (1,6) = 2
Def Use Paths (1,6) = 1
Priority (1,6) = 0.17

du- pair: y(4,8)
ControlPreds (start,4) =2
ControlPreds (4,8) =0
DataFlowPaths(4,8) = 3
Def UsePaths(4,8) = 2
Priority (4,8) = 0.23

du -pair: z (2,3)
ControlPreds (start,2) = 1
ControlPreds (2,3) = 0
DataFlow Paths (2,3) = 5
Def UsePaths (2,3) = 1
Priority (2,3)) = 0.10

du- pair: x (5,6)
ControlPreds (start,5) =3
ControlPreds(5,6) = 0
DataFlowPaths (5,6) = 1
Def UsePaths (5,6) = 1
Priority (5,6) = 0.25

du- pair: y(7,8)
ControlPreds (start,7) = 3
ControlPreds (7,8) =0
DataFlowPaths (7,8) = 1
Def UsePaths (7,8) = 1
Priority (7,8) = 0.25

du- pair: z (9,3)
ControlPreds (start,,9) = 2
ControlPreds (9,3) = 0
DataFlow Paths (9,3) = 2
Def UsePaths (2,3) = 2
Priority (2,3) =0.33

Figure 2: Computing Priorities.

Data Flow Paths

$$DataFlowPaths(src, dest) = \sum_{p \in Pred(dest)} DataFlowPaths(src, p)$$

Def Use Paths

$$DefUsePaths(def_v, use_v) = \sum_{\substack{p \in Pred(use_v) \\ where\ v \notin Kill(p)}} DefUsePaths(def_v, p)$$

Control Predicates

$$ControlPredicates(src, dest) = Predicates[dest]$$

$$Predicates[n] = \begin{cases} 1 + MAX_{(p,n) \in Pred(n) \cap REACH(src,dest)} Predicates[p] & n \in AllControlAnc(dest) \\ MAX_{(p,n) \in Pred(n) \cap REACH(src,dest)} Predicates[p] & otherwise \end{cases}$$

$$REACH(src, dest) = \{n : n\ is\ part\ of\ a\ path\ from\ src\ to\ dest\}$$

Control Dependences

$$AllControlAnc(n) = ImmControlAnc(n) \cup \bigcup_{x \in ImmControlAnc(n)} AllControlAnc(x)$$

$$ImmControlAnc(n) = \{x : \exists s \in Succ(x)\ st\ n \in PostDom(s)\ and\ x \notin PostDom(x)\}$$

$$PostDom(n) = \{n\} \cup \bigcap_{s \in Succ(n)} PostDom(s)$$

Figure 3: Data Flow Equations for Computing Priorities.

1. **Algorithm** *ComputePriorities*()
2. Given an acyclic control flow graph.
3. *ComputeDataFlowPaths()*
4. *ComputeDefUsePaths()*
5. *ComputeControlDependences()*
6. **for each** node def-use pair (def, use) {
7. $Priority(def, use) = \frac{1}{1+ControlPreds(start,def)+ControlPreds(def,use)} \times \frac{DefUsePaths(def,use)}{DataFlowPaths(def,use)}$
8. }
9. **end***ComputePriorities*

1. **Algorithm** *ComputeDataFlowPaths*()
2. **for each** statement node n { $PathsTo[n] \leftarrow \{(n,1)\}$ }
3. $Worklist \leftarrow$ all nodes such that predecessors appear earlier than the successors
4. **for each** node $n \in Worklist$ {
5. $PathsTo[n] \leftarrow \{(s, sum) : sum = \sum_{(s,num_p) \in PathsTo[p],\ where\ p \in Pred(s)} num_p\} \bigcup \{(n,1)\}$
6. }
7. **for each** pair of nodes $(src, dest)$ in the control flow graph {
8. $DataFlowPaths(src, dest) = \begin{cases} num & (src, num) \in PathsTo[dest] \\ 0 & otherwise \end{cases}$
9. }
10. **end***ComputeDataFlowPaths*

1. **Algorithm** *ComputeDefUsePaths*()
2. **for each** statement node n {
3. $DefsIn[n] \leftarrow \phi$
4. $DefsOut[n] \leftarrow \{(n,1)\}$
5. }
6. $Worklist \leftarrow$ all nodes such that predecessors appear earlier than the successors
7. **for each** node $n \in Worklist$ {
8. $DefsIn[n] \leftarrow \{(s, sum) : sum = \sum_{(s,num_p) \in DefsOut[p],\ where\ p \in Pred(s)} num_p\}$
9. $DefsOut[n] \leftarrow \{(s, num) : (s, num) \in DefsIn[n]\ and\ Def(s) \neq Def(n)\} \bigcup \{(n,1)\}$
10. }
11. **for each** def-use pair (def, use) {
12. $DefUsePaths(def, use) = num$, such that $(def, num) \in DefsIn[use]$.
13. }
14. **end***ComputeDefUsePaths*

1. **Algorithm** *ControlPredicates*(*src*,*dest*)
2. $REACH(src, dest)$ = { n: n is a part of a path from *src* to *dest* }
3. $Worklist \leftarrow REACH(src, dest)$ excluding *src* and including *dest*
4. such that predecessors appear earlier than the successors.
5. $Predicates[src] = 0$
6. **for each** node $n \in Worklist$ {
7. $Predicates[n] = MAX_{p \in Pred(n) \cap REACH(src,dest)}(Predicates[p])$
8. **if** $n \in AllControlAnc(dest)$ { $Predicates[n] = Predicates[n] + 1$ }
9. }
10. return(Predicates[dest])
14. **end***ControlPredicates*

Figure 4: Computing Priorities for Def-Use Pairs in Acyclic Graphs.

```
1. Algorithm ComputeControlDependences()
2.     Worklist ← all nodes such that predecessors appear earlier than the successors
3.     for each statement node n { PostDom[n] = {n} }
4.     for each node n ∈ Worklist {
5.             PostDom[n] = {n} ∪ ⋂_{s∈Succ[n]} PostDom[s]
6.     }
7.     for each statement node n { ImmControlAnc[n] = φ }
8.     for each node n ∈ Worklist {
9.             for each node s ∈ Succ(n) {
10.                    for each node x such that x ∈ PostDom(s) and x ∉ PostDom(n) {
11.                            ImmControlAnc[x] = ImmControlAnc[x] ∪ {n}
12.    }}}
13.    for each statement node n { ControlAnc[n] = ImmControlAnc[n] }
14.    for each node n ∈ Worklist {
15.            for each node s such that n ∈ ControlAnc[s] {
16.                    ControlAnc[s] = ContrlAnc[s] ∪ ControlAnc[n]
17.    }}
18. end ComputeControlDependences
```

Figure 5: Computing Priorities for Def-Use Pairs in Acylic Graphs Continued.

In order to handle programs with loops we transform cyclic graphs into acyclic graphs following which the above analysis is applied. The transformations are developed to establish the data flow paths along which various def-use pairs involving a loop can be established. Consider the example in Figure 6(i) which contains a repeat loop. Since def-use pairs can be established within a loop along a path that traverses the loop, we unroll the loop once and establish these def-use pairs. As shown in Figure 6(ii), the def-use pair from 3 to 3′, established across the loop back edge in the original flow graph, is established by the loop body duplication. Consider the def-use pair from statement 1 before the loop to statement 4 after the loop. This def-use pair is established if we do not iterate through the loop but otherwise it is not established. The corresponding data flow path is established in the transformed graph in Figure 6(ii) by adding the edge from 2 to 4. Finally consider the def-use pair from statement 3 in the loop to statement 4 after the loop. This pair is established by adding the edge from 2′ to 4 in Figure 6(ii). Thus, in this example by unrolling the loop once and adding appropriate edges we establish the necessary data flow paths.

Next we present the general loop unwinding transformations based upon the principles illustrated above. Consider the flow graph in Figure 7(i) which contains a single *repeat* loop. In the flow graph representation the node labeled head represents the entry to the loop and tail represents a dummy node that marks the exit of the loop. Duplicating the loop once, as shown in Figure 7(i), is sufficient to ensure the propagation of data flow information from any node in the loop to every node in the loop. As we can see this

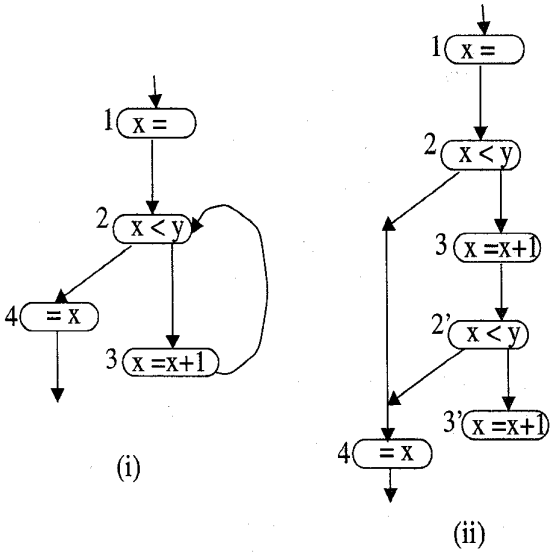


Figure 6: Counting Data Flow Paths and Def-Use Paths in Cyclic Graphs: (i) A Cyclic Flow Graph; and (ii) Corresponding Acyclic Flow Graph.

process creates all static paths present in the original control flow graph. A *while* loop is handled similarly as shown in Figure 7(ii).

The loop unwinding transformations described above create two copies of each statement in the loop. When considering def-use pairs involving these statements, it is important to consider the appropriate copy of a statement. The rules for selecting the appropriate copy of a statement for different kinds of def-use pairs involving a statement in the loop are summarized below.

- If the definition is encountered before entering the repeat/while loop and the use is inside the repeat/while loop body, then the first copy of the use statement must be considered.
- If the definition is encountered inside the loop body and the use is encountered after exiting the loop, then for a repeat loop the second copy of the definition statement must be considered and for the while loop the first copy of the definition statement must be considered.
- If both the definition and the use statements belong to the loop body then we must consider whether the def-use pair is established within a single loop iteration or across loop iterations. In the former case the first copies of the definition and use statements are considered and in the latter case the first copy of the definition and the second copy of the use are considered.

Consider the nested loops shown in Figure 8(i) where the head and tail of each loop is directly connected by an edge. First we handle the outermost loop by replicating the loop body and connecting the tail of one copy with the head of the newly created copy. This process creates the static paths solely due to the outermost loop. Next we must create the static paths for the inner loop by applying the transformation from Figure 7(i) which results in an acyclic graph. In the previous case the exit of the loop is directly connected by the back edge to the loop entry. In some situations, the loop may contain a sequence of back edges that connects the exits of loops to the entries of loops (see Figure 8(ii)). The loop created by the edge from t_1 to h_1 is eliminated first by creating the graph shown in Figure 8(ii). The transformation given in Figure 7(i) will be applied next to eliminated the loop back edge from t_2 to t_1 to obtain an acyclic graph.

An overall algorithm that converts a structured cyclic flow graph into an acyclic flow graph essentially selects the order in which the four transformations discussed in this section are applied. The application of unwinding transformations is preceded by the detection of loops. We identify the largest outermost loop and depending upon its structure use an appropriate transformation to unwind the loop. The unwinding is

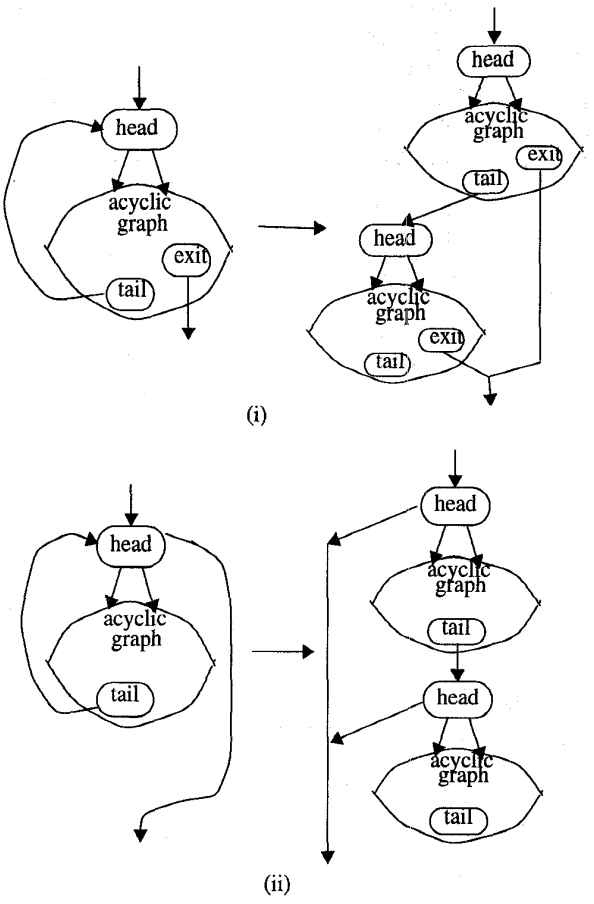


Figure 7: Loop Unwinding Transformations: (i) Unwinding a Repeat Loop; (ii) Unwinding a While Loop.

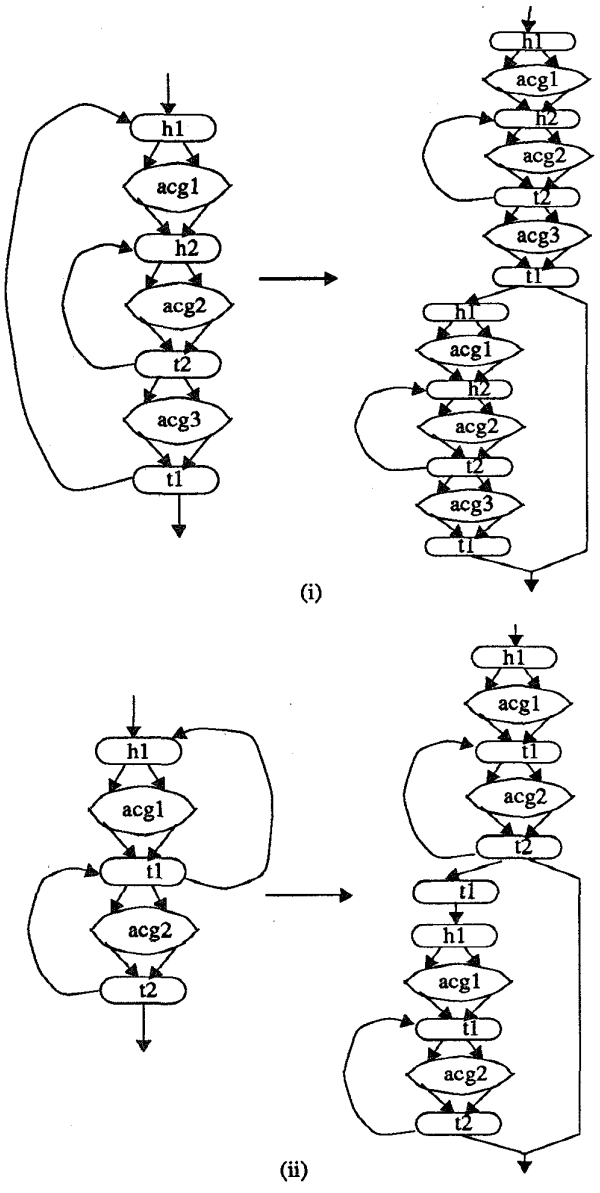


Figure 8: Unwinding Nested Loops: (i) CShell Loops: Head and Tail Connected Directly; (ii) Climbing Loops: Head and Tail Connected Indirectly.

carried out by the function *Unwind* in Figure 9. After the graph has been converted to an acyclic graph we apply the previously presented algorithm to compute the priorities.

1. **Algorithm** *Unwind*()
2. Given a structured flow graph.
3. Identify all loops in the program
4. **while** ∃ a backedge that has not been removed {
5. Select backedge B corresponding to an
6. outermost loop $L = (V_L, E_L)$ in the
7. control flow graph.
8. Remove B by applying one of the following
9. transformations to L:
10. Transf. of Single Loop, Repeat Structure;
11. Transf. of Single Loop, While Structure;
12. Transf. of Cshell Loop; or
13. Transf. of Climbing Loop.
14. }
15. **end** *Unwind*

Figure 9: Applying Loop Unwinding Transformations.

3 Priority based Regression Testing

In the software maintenance phase, program changes may be performed to correct newly discovered errors. Following the correction of errors we must perform regression testing during which only a selected subset of def-use pairs are tested. These def-use pairs are the ones that are effected by the program changes. The testing of some of the effected def-use pairs may require new test cases to be generated while for other def-use pairs existing test cases may suffice. Thus, priority based testing can also be used during regression testing. In addition to the characteristics used earlier in computing priorities, we also consider the number of program changes that affect a particular def-use pair. The greater the number of changes that affect a def-use pair the greater is the priority assigned to it. Algorithms for identifying the def-use pairs effected by a program change is found in [5]. The priorities are more formally defined below.

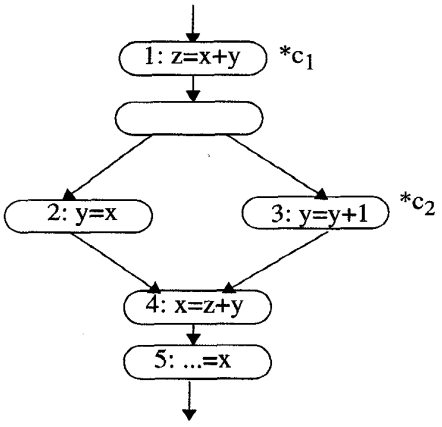
Definition 5: A def-use pair (def, use) is *effected* by a program change if either it is created due to the program change or the computation of the value at the definition def is effected directly or indirectly by a program change. The set of changes that affect a def-use pair (def, use) is denoted by $AffectingChanges(def, use)$.

Definition 6: The *priority* of a def-use pair (def, use) during regression testing, denoted by $RegrPriority(def, use)$, is defined as follows:

$$RegrPriority(def, use) = Priority(def, use) \times \mid AffectingChanges(def, use) \mid .$$

From the above definition it is clear that if a def-use pair is not effected by a program change its priority will be zero indicating that it does not need to be retested. However, the priority will rise with the number of changes that affect a def-use pair and in fact the priority of a def-use pair can exceed one and is bounded only by the number of changes made to a program.

The example in Figure 10 illustrates the modified definition for priority. Let us assume that changes have been made in the program to statements 1 and 3. The code fragment contains four def-use pairs. The def-use pair of variable x from statement 4 to statement 5 is given a higher priority than the remaining def-use pairs because the former pair is effected by both program changes while the remaining effected def-use pairs are effected by exactly a single program change.



du-pairs	Affecting Changes	RegrPriority
z (1,4)	{ c_1 }	Priority(1,4)
y (3,4)	{ c_2 }	Priority(3,4)
x (4,5)	{ c_1, c_2 }	2*Priority(4,5)
y(2,4)	{}	0

Figure 10: Priorities for Regression Testing.

4 Conclusion

In this paper we presented an approach for improving the effectiveness of the effort spent on the data flow testing of a program. Our approach is based upon prioritizing the def-use pairs according to the expected cost of testing the def-use pair, which is measured in terms of the difficulty in finding a test case for the pair. In case of regression testing the priorities are assigned to capture not only the difficulty in test case generation but also the likelihood that an error introduced by a program change will be caught by the test case. We are currently investigating ways to develop more effective testing methodologies that integrate path testing with def-use testing. Since the priorities of def-use pairs are based upon analysis of paths from a definition to a corresponding use, our approach is suitable for enabling the integration of path testing with data flow testing.

References

[1] H. Agrawal, "Dominators, Super Blocks, and Program Coverage," *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 25-34, Jan. 1994.

[2] D. Alberts, "The Economics of Software Quality Assurance," *Proc. of the AFIPS 1976 National Computer Conference*, Vol. 45, pp. 433-442, 1976.

[3] R. Gupta, "Generalized Dominators and Post-Dominators," *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, pages 246-257, Jan. 1992.

[4] R. Gupta and M.L. Soffa, "Employing Static Information in the Generation of Test Cases," *Journal of Software Testing, Verification and Reliability*, Vol. 3, No. 1, pages 29-48, December 1993.

[5] R. Gupta, M.J. Harrold, and M.L. Soffa, "An Approach to Regression Testing using Slicing," *Proc. of the Conference on Software Maintenance*, pages 299-308, Orlando, Florida, November 1992.

[6] M.J. Harrold and M.L. Soffa, "An Incremental Approach to Unit Testing During Maintenance," *Proc. of the Conference on Software Maintenance*, pages 362-367, 1988.

[7] K.W. Kennedy, "Node Listings Applied to Data Flow Analysis," *Proc. of the 2nd ACM Symposium on Principles of Programming Languages*, pages 10-21, 1975.

[8] R. Kramer, R. Gupta, and M.L. Soffa, "The Combining DAG: A Technique for Parallel Data Flow Analysis," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 8, pages 805-813, August 1994.

[9] G. Myers, "The Art of Software Testing," Wiley, New York, 1979.

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

[11] 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.