A Methodology for Controlling the Size of a Test Suite

MARY JEAN HARROLD
Clemson University
and
RAJIV GUPTA and MARY LOU SOFFA
University of Pittsburgh

This paper presents a technique to select a representative set of test cases from a test suite that provides the same coverage as the entire test suite. This selection is performed by identifying, and then eliminating, the redundant and obsolete test cases in the test suite. The representative set replaces the original test suite and thus, potentially produces a smaller test suite. The representative set can also be used to identify those test cases that should be rerun to test the program after it has been changed. Our technique is independent of the testing methodology and only requires an association between a testing requirement and the test cases that satisfy the requirement. We illustrate the technique using the data flow testing methodology. The reduction that is possible with our technique is illustrated by experimental results.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**] Distribution and Maintenance—*version control*; D.2.10 [**Software Engineering**]: Design—*methodologies*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software maintenance*

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: Hitting set, maintenance, regression testing, software maintenance, software engineering, test suite reduction, testing.

1. INTRODUCTION

To accommodate testing during the many stages of a program's lifetime, a suite of test cases is typically developed and stored with the program. Since test cases in the existing test suite can often be used to test a modified program, the test suite is used for retesting. However, if the test suite is

An earlier version of this paper appeared in abridged form in *Proceedings of the Conference on Software Maintenance 1991*
This work was partially suppported by the National Science Foundation under grants CCR-9109531 to Clemson University, NSF Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh, and grant CCR-9109089 to the University of Pittsburgh
Authors' addresses: M. J. Harrold, Department of Computer Science, Clemson University, Clemson, SC 29634-1906; R. Gupta and M. L. Soffa, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

inadequate for retesting, new test cases may be developed and added to the test suite. Thus, changes to an evolving program cause the size of the test suite to grow, making efficient test suite management desirable. A reduction in the size of the test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software. Managing the stored test suite includes both designing new test cases and eliminating unnecessary test cases. New test cases are designed wherever they are required to test the changed program. However, in practice, test cases are not removed from the test suite, since it is usually difficult to identify those test cases that are no longer necessary.

Unnecessary test cases include both obsolete and redundant test cases. A change in a program causes a test case to become *obsolete* by removing the reason for the test case's inclusion in the test suite. A test case is *redundant* if other test cases in the test suite provide the same coverage of the program. Thus, because of obsolete and redundant test cases, the size of the test suite continues to grow unnecessarily as software changes are made. Determining a minimum set of test cases that provides the same coverage of the changed or affected parts of the program is desirable. One technique [14] uses a dynamic analysis of the program to reduce the number of test cases. This technique executes the program with test cases from the test suite until the structural coverage criterion is satisfied. Any remaining unexecuted test cases are obsolete and can be eliminated. The ad hoc way in which the test cases are chosen impacts on the number of redundant test cases that are found. Further, this technique applies only to structural coverage and not to testing criteria in general.

This paper presents a new reduction technique that assists in a number of stages of testing deterministic programs. To perform the reduction, we develop a heuristic that selects a representative set of test cases that is a subset of a test suite, but still provides the desired testing coverage[1] of the program. We use coverage to mean *any* method of completeness with respect to a test selection criterion [2]. The reduction technique requires an association between the test cases and the testing requirements of the program, but is independent of the test selection criteria and can be applied if this association can be made. The reduction technique can also accommodate test suites that reflect more than one test selection criteria. The reduction can be performed on the entire, stored test suite or on a test suite consisting of those test cases that test changed or affected parts of a program. We implemented our technique by incorporating it into a data flow testing system [8] that is part of the Free Software Foundation's GNU CC compiler.© (Copyright (C) 1987, 1989 Free Software Foundation, Inc., 675 Mass Avenue, Cambridge, MA 02139.) Our experiments include using the technique during initial program development, after implementation-type and functional changes are made to the program. Our results demonstrate the reduction in the size of the test suite that can be achieved by our technique.

[1] We use *coverage* to mean any metric of completeness with respect to a test selection criterion [2] and thus, it applies to any testing methodology.

In the next section, we discuss approaches to retesting after program changes and illustrate the usefulness of our technique. Section 3 details the test suite reduction algorithm as it applies to a single test selection criterion and an example is given. Section 4 demonstrates our methodology as applied to data flow testing. In Section 5, we discuss the way in which we handle test suites based on more than one test selection criteria and thus, illustrate the generality of our technique. Our experimental results are presented in Section 6 and concluding remarks are given in Section 7.

2. TEST SUITE MANAGEMENT

For a particular program, a test selection criterion translates into a set of *test case requirements*, whose satisfaction provides the desired measure of completeness with respect to that criterion. For each test case requirement, an *associated testing set* consists of the subset of the test suite that satisfies the requirement. Our test suite reduction technique uses both the test case requirements and their associated testing sets. *Black box* or functional testing uses the program specifications to identify the test case requirements. The associated testing sets are determined during test suite development. Whenever a test case is developed for a specification, the user includes the test case in the associated testing set for any other specification that it also tests. *White box* or program-based testing uses the components of the program to derive the test case requirements and the associated testing sets are determined by considering the execution paths of the test cases. The program is instrumented to record the execution path of a test case that is then used to associate test cases with program-based requirements in the program.

A program modification may cause a change in a program's test case requirements: new requirements may be added or existing requirements may be deleted/modified. Although some existing test cases in the test suite may retest the modified software, the change in test case requirements may require new test cases and may also allow unnecessary test cases to be eliminated. There are two basic approaches to retesting after program changes: incremental [7, 10, 11, 16, 18] and retest-all [13]. Both of these approaches can benefit from our reduction technique.

An incremental approach to testing after program changes can use our reduction technique in two ways. First, it can use the technique to find a reduced set of test cases for the retesting. With incremental testing, an analysis uses information saved from previous testing sessions to determine the effects of program modifications. The analysis updates the test case requirements to reflect the changed program and identifies a subset of the test suite for the retesting. There may be redundancy in this subset since several test cases may test the same changed parts of the program. Although it is desirable to eliminate this redundancy and reduce the amount of retesting, existing incremental methods do not identify a smaller set of test cases that provides the required coverage of the modified program. Thus, our

reduction technique can be used to reduce the retesting by eliminating these redundant test cases.

The second way that an incremental approach can use our reduction technique is to reduce the size of the stored test suite. The test suite is continually updated so that it always reflects those test cases that satisfy the test case requirements. After executing the program with the test cases identified for retesting, an incremental approach determines unsatisfied test case requirements and reports them to the user. New test cases are developed for these unsatisfied test case requirements and added to the test suite. Any test case that becomes obsolete, because of the removal of the only testing requirement associated with it, is eliminated. Although this approach does eliminate test cases that no longer satisfy a test case requirement, it does not eliminate redundant test cases. New test cases that are added to the test suite may cause other test cases in the test suite to become redundant, in that they satisfy the same test case requirements. Current incremental testing techniques do not eliminate redundant test cases. Our reduction technique can be used to eliminate redundancy and reduce the size of the stored test suite.

A retest-all approach tests modified programs without identifying a subset of the test suite that can be used for the retesting. Thus, the program is executed with all test cases in the test suite. More test cases are added until all test case requirements are satisfied. In the retest-all approach, the associated testing sets are not updated to reflect the removal of test case requirements and thus, obsolete test cases remain in the test suite. Additionally, there is no attempt to remove redundant test cases. The test suite can become quite large since unnecessary test cases are not removed and new test cases are added. Our reduction technique can be used to eliminate both obsolete and redundant test cases from the test suite.

3. TEST SUITE REDUCTION

We state the problem of selecting a representative set of test cases that provides the desired testing coverage of a program or part of a program as follows:

Given: A test suite TS, a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired testing coverage of the program, and subsets of TS, $T_1, T_2, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Problem: Find a representative set of test cases from TS that satisfies all of the r_i's.

The r_i's can represent either all of the program's test case requirements or those requirements related to program modifications. A representative set of test cases that satisfies the r_i's must contain at least one test case from each T_i. Such a set is called a *hitting* set of the group of sets $T_1, T_2, \ldots, T_n$. A maximum reduction is achieved by finding the smallest representative set of

test cases. However, this subset of the test suite is the minimum cardinality[2] hitting set of the T_i's and the problem of finding the minimum cardinality hitting set is NP-complete [4]. Therefore, since we are unaware of any approximate solution to the problem, we develop a heuristic [5, 6] to find a representative set that approximates the minimum cardinality hitting set.

The heuristic first includes all test cases that occur in single element T_i's in the representative set and marks all T_i's containing any of these test cases. Then, all unmarked T_i's of cardinality two are considered. Repeatedly, the test case that occurs in the maximum number of T_i's of cardinality two is chosen and added to the representative set. Again, all unmarked T_i's containing these test cases are marked. This process is repeated for T_i's of cardinality $3, 4, \ldots, max$, where max is the maximum cardinality of the T_i's. We consider the T_i's with lesser cardinalities first because they contain fewer test cases from which to choose. At any given time, only those T_i's from which no element has yet been chosen are considered. When examining the T_i's of size n, there may be a tie because several test cases occur in the maximum number of T_i's of that size. In this case, the heuristic examines the unmarked T_i's with cardinality $(n + 1)$ for those test cases that were involved in the tie. The test case that occurs in the maximum number of T_i's of cardinality $(n + 1)$ is chosen. If a decision cannot be made, the T_i's with greater cardinality are examined and finally a random choice is made.

For example, in Table I, the test suite TS consists of test cases, t_i, the test case requirements, REQ_i, and the associated testing sets, T_i. The heuristic first adds test case t_5 to the representative set since T_2 is the only singleton T_i. REQ_1 and REQ_2 are marked as being satisfied since t_5 is associated with each of them. Then, we consider unmarked T_i's of cardinality two (i.e., T_4, T_5, and T_6). Each of the test cases t_3 and t_4 appears in one of these T_i's while each of test cases t_1 and t_6 appears in two of those T_i's. Since there is a tie between test cases t_1 and t_6 for the maximum, we continue processing with unmarked T_i's of the next higher cardinality. Thus, T_3 and T_7 are considered next. We only use the test cases involved in the tie to compute the maximum for cardinality 3. Test case t_1 appears in T_3 while test case t_6 appears in neither of the T_i's. Thus, test case t_1 is chosen and added to the representative set. T_3, T_5, and T_6 are marked since they contain t_1. Processing continues with T_4, the only unmarked r_i of cardinality 2. Again there is a tie between t_3 and t_6 causing T_i's of cardinality 3 to be examined. Test case t_3 appears in T_7, and thus, it is added to the representative set, which allows the remaining T_i's (T_4, T_7, and T_8) to be marked. The resulting representative set is $\{t_1, t_3, t_5\}$. For the test plan update problem, t_2, t_4, t_6, and t_7 can be eliminated from the test suite. For the test selection problem, only test cases t_1, t_3, and t_5 must be rerun to satisfy the test case requirements.

The algorithm, *ReduceTestSuite*, that implements this heuristic is given in Figure 1. In step 1, algorithm *ReduceTestSuite* initializes and preprocesses the T_i's. First, the maximum cardinality, MAX_CARD, of the input sets is

[2] The *cardinality* of a finite set is the number of elements in the set.

Table I. Test Suite TS Consists of Test Cases t_i, Testing Requirements are REQ_i and Associated Testing Sets are T_i

i	r_i	T_i
1	REQ_1	$\{t_1, t_5\}$
2	REQ_2	$\{t_5\}$
3	REQ_3	$\{t_1, t_2, t_3\}$
4	REQ_4	$\{t_3, t_6\}$
5	REQ_5	$\{t_1, t_4\}$
6	REQ_6	$\{t_1, t_6\}$
7	REQ_7	$\{t_3, t_4, t_7\}$
8	REQ_8	$\{t_2, t_3, t_4, t_7\}$

determined and the cardinality currently being processed, CUR_CARD, is initialized to one. Next, *ReduceTestSuite* initially forms the representative set, RS, by taking the union of all of the T_i's that are single element sets. All T_i's containing some element in the initial RS are marked; marked sets are not reprocessed. Then, in Step 2, *ReduceTestSuite* determines the remainder of RS. Unmarked T_i's with increasing cardinality are examined. A list, LIST, is constructed that consists of those test cases in the T_i's with cardinality equal to the current cardinality, CUR_CARD. A new test case, NEXT_TEST, is chosen from test cases in LIST and added to RS. Subsequently, all remaining unmarked T_i's containing NEXT_TEST are marked as represented in RS. The function, *SelectTest*, is used to determine NEXT_TEST, the next test case to add to RS. *SelectTest* recursively examines the T_i's of cardinality SIZE for test cases contained in the greatest number of the T_i's. The array COUNT is used to store the number of T_i's of cardinality SIZE containing the t_j's in LIST. If several T_j's are found, the T_i's of the next higher cardinality are examined to select a single test case from these test cases. *ReduceTestSuite* repeats this process until either a single test case is found or one is chosen at random. A random choice is made if no higher cardinality sets remain to be examined. The Boolean variable MAY_REDUCE is set whenever one of the sets with cardinality MAX_CARD is marked. Whenever MAY_REDUCE is set, MAX_CARD is reinitialized to the highest cardinality of the remaining sets. Resetting MAX_CARD is an optimization that may reduce the time to select the next test case.

We analyze the worst case run-time of *ReduceTestSuite* to demonstrate that it is efficient and therefore suitable to incorporate into a test suite management methodology. Let n denote the number of associated testing sets T_i, nt denote the number of test cases t_i, and MAX_CARD, the maximum cardinality of the groups of sets. *ReduceTestSuite* involves two main steps: (1) computing the number of occurrences of various test cases in sets of varying cardinality and (2) selecting the next test case to add to the representative set. These steps are performed repeatedly until a representative set is found. Computing the number of occurrences of various test cases in sets of varying cardinality takes $O(n^*\text{MAX\_CARD})$ time since there are n sets and all elements of these sets are examined once. Selecting the next test case to

```
algorithm ReduceTestSuite

input     T1, T2, ···, Tn: associated testing sets for r1, r2, ···, rn respectively, containing test cases from t1, t2, ···, tnt
output    RS: a representative set of T1, T2, ···, Tn
declare   MAX_CARD, CUR_CARD: 1 .. nt
          LIST: list of ti's
          NEXT_TEST: one of t1, t2, ···, tnt
          MARKED: array[1..n] of boolean, initially false
          MAY_REDUCE: boolean
          Max(): returns the maximum of a set of numbers
          Card(): returns the cardinality of a set
begin
/* Step 1: initialization */
     MAX_CARD = Max(Card(Ti))                          /* get the maximum cardinality of the Ti's */
     RS = ⋃i Ti, Card(Ti) = 1                          /* take union of all single element Ti's */
     foreach Ti such that Ti ⋂ RS ≠ Φ do MARKED[i] := true   /* mark all Ti containing elements in RS */
     CUR_CARD := 1                                     /* consider single element sets first */
/* Step 2: compute RS according to the heuristic for sets of higher cardinality */
     loop
        CUR_CARD := CUR_CARD + 1                       /* consider the sets with next hier cardinality */
        while there are Ti such that (Card(Ti) = CUR_CARD and not Marked[i] do
                                                       /* process all unmarked sets of current cardinality */
           LIST := all tj ∈ Ti where Card(Ti) = CUR_CARD and not Marked[i]
                                                       /* all tj in Ti of size CUR_CARD */
           NEXT_TEST := SelectTest(CUR_CARD,LIST)      /* get another tj to include in RS */
           RS := RS ⋃ {NEXT_TEST}                      /* add the test to RS */
           MAY_REDUCE := false
           foreach Ti where NEXT_TEST ∈ Ti do
              MARKED[i] := true                        /* mark Ti containing NEXT_TEST */
              if Card(Ti) = MAX_CARD then MAY_REDUCE := 1
           endfor
           if MAY_REDUCE then                          /* try to reduce MAX_CARD */
              MAX_CARD := Max(Card(Ti)), for all i where MARKED[i] = false
        endwhile
     until CUR_CARD = MAX_CARD
end ReduceTestSuite.

- - - - - - - - -

function SelectTest(SIZE, LIST)
/* this function selects the next ti to be included in RS */

declare   COUNT: array[1 .nt]
begin
     foreach ti in LIST do compute COUNT[ti], the number of unmarked Tj's of cardinality SIZE containing ti
     Construct TESTLIST consisting of tests from LIST for which COUNT[i] is the maximum
     if Card(TESTLIST)=1 then return(the test case in TESTLIST)
     elseif SIZE = MAX_CARD then return(any test case in TESTLIST)
     else return(SelectTest(SIZE+1, TESTLIST))
end SelectTest.
```

Fig. 1 Algorithm *ReductTestSuite* for finding a representative set from a group of sets.

be included in the representative set requires examining the counts associated with each test case. This step takes at most $O(nt * \text{MAX\_CARD})$ time. Selecting a test case and recomputing the counts is repeated at most n times since after selecting a test case at least one additional test case is covered by the representative set. Therefore, the overall run-time of *ReduceTestSuite* is $O(n(n + nt)\text{MAX\_CARD})$.

Although the above analysis provides the worst-case run-time of *ReduceTestSuite*, our experiments demonstrate that in practice it executes much faster. We first recorded the actual execution times of *ReduceTestSuite* for a set of test programs using the associated testing sets obtained during program testing. Then, for each test program and its testing requirements, we constructed a second set of associated testing sets to force *ReduceTestSuite* to perform poorly and recorded the actual execution times of *ReduceTestSuite* with these associated testing sets. For a set of test case requirements, $r_1, r_2, \ldots, r_n$ and a set of test cases, $t_1, t_2, \ldots, t_{nt}$ the associated testing sets are formed as follows:

$$T_i \text{ for } r_i \text{ is } \{t_{(i-1)\text{MOD } nt+1}, t_{(i)\text{MOD } nt+1}\} \text{ for } i = 1, 2, \ldots, n-1$$
$$T_n \text{ for } r_n \text{ is } \{t_1, t_2, \ldots, t_{nt}\}$$

Thus, for a test program having five test case requirements and four test cases, we construct the following associated testing sets:

$$T_1 = \{t_1, t_2\}$$
$$T_2 = \{t_2, t_3\}$$
$$T_3 = \{t_3, t_4\}$$
$$T_4 = \{t_4, t_1\}$$
$$T_5 = \{t_1, t_2, t_3, t_4\}.$$

Since this construction of T_i's contains little overlap in the subsets, a minimum number of r_i are satisfied by each test case. Furthermore, since one of the T_i's is the entire test suite and a tie will occur in all other subsets, the computation will consider the maximum cardinality at least once. Table II compares the execution time of *ReduceTestSuite* on the associated testing sets obtained during program testing with the worst-case scenario found by constructing the associated testing sets from the same test suite. In all cases, *ReduceTestSuite* performs better on the actual associated testing sets found during testing than the worst-case constructed sets; in most cases, it performs much better on the actual associated testing sets.

4. APPLICATION TO DATA FLOW TESTING

We incorporated the representative set algorithm into a data flow testing system [8]. Data flow testing is a methodology that uses the data dependencies in a program to guide the selection of the test cases [3, 12, 15]. Data flow analysis determines the relationships between *definitions* of variables and *uses* of the same variable. Definitions of variables occur in statements where a variable gets a value, such as assignment statements and input statements. Uses of variables occur where a variable's value is fetched, such as output statements, conditional statements and the right-hand side of assignment statements. A definition that reaches a use forms a definition–use pair. We illustrate the reduction technique using the Rapps and Weyuker data flow criterion [17], although it can also be applied to other criteria. For the

Table II. Run-Times for *ReduceTestSuite* for Actual and Constructed Associated Testing Sets

procedure	*test cases*	*actual associated testing sets*	*constructed associated testing sets*
trityp	16	1.50	9.28
atof	2	.07	.13
getop	4	.28	.80
calc	7	.23	.60
qsort	5	.10	.30
trityp2	19	.27	2.35
sqroot	6	.07	.35
sqroot2	6	.10	.41
sqroot3	6	.25	.62
sqroot4	5	.08	.20
sqroot5	6	.10	.25

"all-uses" criterion, definition-use pairs are computed both within individual procedures [1] and across procedure boundaries [9]. The required definition-use pairs are identified as the test case requirements and test cases that satisfy the definition-use pairs are developed. A test case satisfies a particular definition–use pair if a subpath from the definition to the use, with no intervening redefinition of the variable, is traversed during the program's execution with the test case as input. To determine if a test case satisfies a particular definition-use pair, the program is instrumented so that it outputs the execution path of a test case. Then, an *acceptor* takes both a definition–use pair and the execution path and determines whether the test case satisfies the path. To determine the associated testing sets, the acceptor is run with each of the testing requirements and each of the test cases as input. *ReduceTestSuite* is used to provide a reduction in the number of test cases to rerun to validate a changed program and to control the size of the test suite.

To illustrate the *ReduceTestSuite* algorithm as it applies to the test update problem, consider the flow graph for a simple program segment given in Figure 2. Statements of the form "$X :=$ " represent definitions of X while statements of the form "$:= X$" represent uses of X. Data flow analysis is performed on the program and the required definition–use pairs are identified. The definition of X in B_1 has uses in B_2, B_3, B_5, and B_6 and the definition of X in B_3 has uses in B_5 and B_6. A definition-use pair is denoted by an ordered pair where the first coordinate is the node in the control flow graph containing the definition and the second coordinate is the node containing the use. The definition–use pairs for variable X are: (B_1, B_2), (B_1, B_5), (B_1, B_6), (B_1, B_3), (B_3, B_5), and (B_3, B_6). Test cases are supplied and accepted until all required definition–use pairs have been satisfied or dismissed as infeasible by the user[3]. A possible set of test cases, their execution paths and the definition–use pair(s) satisfied by each of them is given in Table III.

[3] A path through a program for which no input will cause its execution is said to be *infeasible*.

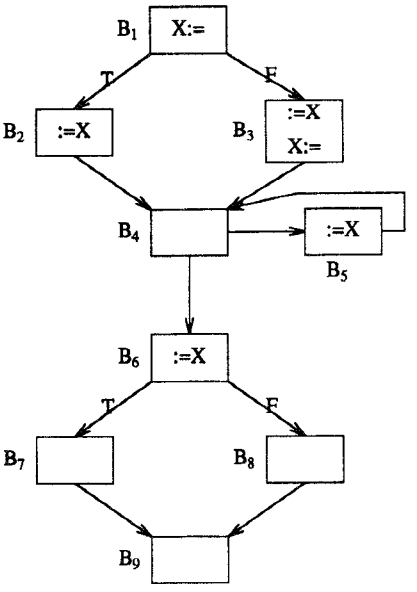

Fig. 2. Control flow graph for partial example program: "$X :=$ " represents a definition of X and " $= X$" represents a use of X.

Table III. Testing Information

test case	execution path	definition–use pair(s)
1	$B_1, B_2, B_4, B_6, B_7, B_9$	$(B_1, B_2), (B_1, B_6)$
2	$B_1, B_2, B_4, B_5, B_6, B_7, B_9$	(B_1, B_5)
3	$B_1, B_3, B_4, B_6, B_8, B_9$	$(B_1, B_3), (B_3, B_6)$
4	$B_1, B_3, B_4, B_5, B_6, B_8, B_9$	(B_3, B_5)

Table III(a). Addendum to Table III, Reflecting New Test Case Requirements and New Test Cases

test case	execution path	definition–use pair(s)
5	$B_1, B_2, B_4, B_6, B_8, B_9$	(B_1, B_8)
6	$B_1, B_3, B_4, B_5, B_6, B_7$	(B_3, B_7)

Test cases are numbered according to the order in which they were supplied for testing.

Now suppose that the program is changed by inserting uses of variable X in statements in nodes B_7 and B_8. The partial control flow graph given in Figure 3 shows the changes that result. Due to the changes, the following definition–use pairs are added (B_1, B_7), (B_1, B_8), (B_3, B_7), and (B_3, B_8). Tests t_1 and t_2 satisfy (B_1, B_7) and tests t_3 and t_4 satisfy (B_3, B_8). However, since no existing test cases can be used for the other two pairs, additional test cases are developed. Two test cases are required since the definitions are on different control paths in the program. Thus, the following information is added to Table III.

Consider a request for a test suite reduction at this point. Inspection of the existing test cases reveals some redundancy. For example, test t_6 also

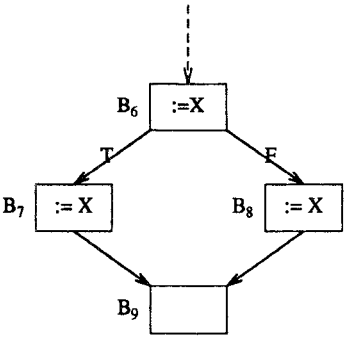

Fig. 3. Control flow graph for partial example program: "X = " represents a definition of X; "$:= X$" represents a use of X.

Table IV. Definition–Use Pairs and Their Associated Testing Sets

definition–use pair	associated testing set
(B_1, B_2)	$\{t_1, t_2, t_5\}$
(B_1, B_5)	$\{t_2\}$
(B_1, B_6)	$\{t_1, t_2, t_5, t_6\}$
(B_1, B_3)	$\{t_3, t_4, t_6\}$
(B_3, B_5)	$\{t_4, t_6\}$
(B_3, B_6)	$\{t_3, t_4, t_6\}$
(B_1, B_7)	$\{t_1, t_2\}$
(B_1, B_8)	$\{t_5\}$
(B_3, B_7)	$\{t_6\}$
(B_3, B_8)	$\{t_3, t_4\}$

satisfies (B_1, B_6) and (B_1, B_3). The associated testing sets for the definition-use pairs are identified and listed in Table IV. *ReduceTestSuite* considers the associated testing sets. First, the singleton sets $\{t_2\}$, $\{t_5\}$, and $\{t_6\}$ are added to the representative sets and all definition–use pairs associated with any of these test cases are marked. Thus, all definition–use pairs except (B_3, B_8) are marked. Then, the remaining sets are inspected and elements from doubleton sets are identified and added to the representative set. Here, only the set $\{t_3, t_4\}$ remains, so t_3 is added to the representative set and (B_3, B_8) is marked. The resulting representative set is $\{t_2, t_3, t_5, t_6\}$, which becomes the new test suite since other test cases in the test suite are eliminated. Thus, two of the six test cases are redundant and can be eliminated without losing the desired testing coverage of the program.

The reduction technique can also be applied to the test selection problem. Suppose that incremental analysis of another change in the modified example program indicates that all existing definition–use pairs for the definition of X in B_1 must be retested. These definition–use pairs are: (B_1, B_2), (B_1, B_3), (B_1, B_6), (B_1, B_7), and (B_1, B_8). The associated testing sets for these definition–use pairs are listed in Table V. The heuristic can be used to identify a representative set for the retesting that still provides the desired coverage of the changed part of the program. Test cases t_2 and t_5 are added to

Table V. Definition–Use Pairs and Their Associated Testing Sets

definition–use pair	associated testing set
(B_1, B_2)	$\{t_2, t_5\}$
(B_1, B_3)	$\{t_3, t_6\}$
(B_1, B_5)	$\{t_2\}$
(B_1, B_6)	$\{t_2, t_5, t_6\}$
(B_1, B_7)	$\{t_2\}$
(B_1, B_8)	$\{t_5\}$

the representative set and definition-use pairs (B_1, B_2), (B_1, B_5), (B_1, B_6), (B_1, B_7), and (B_1, B_8) are marked. Test case t_3 is added to the representative set since either of test cases t_3 or t_6 satisfy definition-use pair (B_1, B_3). Rerunning the program with test cases t_2, t_3, and t_5 provides the desired coverage of the changed program.

5. TEST SUITES REPRESENTING MULTIPLE TEST SELECTION CRITERIA

We have presented our technique for controlling the size of a test suite as if a single test selection criterion were used for the testing. However, typically in practice, a program is validated using test suites that satisfy a combination of different testing methodologies. For our purposes, we classify testing methodologies according to the relationships among the test suites that are used for the testing. We discuss the technique as it applies to two test selection criteria but it can be easily extended to handle a greater number of methodologies. There are three broad classifications of the relationships among test suites: (1) the test suite associated with one methodology is a subset of the other, (2) the test suite of the two methodologies may overlap but one is not contained in the other, and (3) the test suites of the two methodologies are completely independent.

The first classification occurs when combining testing methodologies such that one methodology creates test cases that are also used in the other methodology. For example, consider using both functional and structural testing. Functional testing is usually performed first on a program and test cases are developed and used to test the program specifications. Then, these test cases are used to determine the coverage achieved according to some structural requirements of the testing. Test cases are developed for any structural testing requirements that remain unsatisfied. In this case, the set of functional test cases is a subset of the set of structural test cases. Thus, any reduction of the test suite for the structural testing should not eliminate any of the required functional tests. We apply our algorithm to this combination of functional and structural test suites by first identifying the representative set for the required functional test cases. Then, we use this representative set as the initial representative set for the structural test suite. Those associated testing sets whose intersection with the initial representative set is not empty are marked. The rest of the representative set for the structural test suite is identified by applying the algorithm to the

remaining associated testing set. For this combination of testing methodologies, one representative set is always a subset of the other.

The second classification occurs when combining testing methodologies such that there is a nonempty intersection between the test suites for the two methodologies. Here, the test suites for the two sets of test case requirements have some test cases in common, but one test suite is not necessarily a subset of the other. A representative set is first identified for one of the testing methodologies. Then the intersection of this representative set and the second test suite becomes the initial representative set for the second test suite. Any associated testing sets whose intersection with the initial representative set is nonempty is marked, and the heuristic is applied to the remaining associated testing sets to complete this representative set. Although one of the representative sets may be a subset of the other, in general, this is not the case. However, the two representative sets have some test cases in common.

In the third combination of testing methodologies, the intersection of the test suites associated with the test case requirements is empty. Thus, the representative sets for the two test suites will have no test cases in common. The representative set for each of the test suites is computed independently by applying the heuristic to each of them.

6. EXPERIMENTATION

We incorporated a data flow tester into Free Software Foundation, Inc's GNU CC compiler© [8] and used our algorithm to implement the test suite reduction technique. We performed three separate experiments to determine the possible reduction of the test suite size that our technique can provide for different stages of the software life cycle. Our experiments used two different test selection criteria (1) function test cases based on a program's specifications and (2) "all-uses" data flow testing criterion. In all experiments, the functional test cases were used to test the program and then test cases were developed to satisfy any uncovered definition-use pairs. We described the reduction technique for this type of test suite in Section 5 (the first classification) and our reduction is performed accordingly. We used a team of testers consisting of five graduate students having varying familiarity with both the project and data flow testing (e.g., one of the students implemented the tester while another had no previous knowledge of formal testing procedures). The students were given the programs to be tested and instructions about using the tester. Each member of the testing team performed the experiments individually. We summarize the results of the experiments in Tables VI, VII, and VIII. For each program tested, its number of source lines, number of definition–use pairs, and number of test cases before the reduction is listed. Then, the number of redundant test cases found by the experiment along with the percent of reduction is given.

In Experiment 1, we used our data flow tester to find an "all-uses" adequate test set for a set of small C programs by starting the experiment with a given set of functional test cases. The program descriptions and

Table VI. Experiment 1: Reduction During Program Development

procedure	*source lines*	*du-pairs*	*original test cases*	*redundant test cases*	*%reduction*
trityp	21	39	16	3	18.7
atof	17	63	2	1	50.0
getop	19	33	5	3	60.0
calc	33	3	11	4	36.4
qsort	20	43	4	2	50.0
sqroot	19	13	5	2	40.0

Table VII. Experiment 2: Reduction During Program Maintenance for Performance Improvement

procedure	*source lines*	*du-pairs*	*original test cases*	*redundant test cases*	*%reduction*
trityp2	30	42	13	7	54.6
sqroot2	21	25	6	2	33.3
sqroot3	33	44	5	1	20.0
sqroot4	17	17	7	2	28.6
sqroot5	17	24	5	1	20.0

Table VIII. Reduction During Program Maintenance for Program Enhancements

procedure	*source lines*	*du-pairs*	*original test cases*	*redundant test cases*	*%reduction*
calc2	41	4	80	0	0.0
calc3	60	4	13	4	30.8
calc4	72	4	14	0	0.0
calc5	86	16	18	3	16.7
getop2	27	57	4	1	25.0
getop3	38	69	5	2	40.0

reduction achieved are shown in Table VI. *Trityp* is a program that classifies triangles as to their types, *atof* is the C procedure that converts character data to floating point data, *getop* and *calc* are routines for a calculator program, *qsort* is a quicksort routine, and *sqroot* is a program to calculate the square root. After initial testing, we applied our reduction technique to the resulting test set to find a representative set of test cases that provides the same coverage of the program. This reduced test set replaced the original test set. Our goal here was to identify and remove redundant test cases introduced during program development. In all cases, our algorithm achieved significant reduction in the number of test cases.

In Experiment 2, we made changes to the implementations of two of the programs without changing the program's specification. In *trityp*2, we improved the implementation of the *trityp* program to make it more efficient. We also changed the algorithm used for the *sqroot* program in several ways to get *sqroot*2, *sqroot*3, *sqroot*4, and *sqroot*5. In all cases, we used the

reduced test suite of Experiment 1 to retest, so the test set had already been reduced before we started testing in Experiment 2. With this experiment, we hoped to gain insights into the possible test set reduction that could be achieved during program maintenance when only structural program changes were made. We used the technique described in Section 5 by first reducing the functional test cases and then reducing the rest of the test suite. Table VII gives the results of Experiment 2. Again, in all cases, our technique found redundant test cases, so a reduced test suite results.

In Experiment 3, we changed the function of the *calc* program by adding new features. Since *calc* calls *getop*, both were changed and retested. Here, we were changing both the function and structure of the program. Our objective in this experiment was to determine the amount of redundancy introduced by program enhancements. Table VIII shows the results of our experiments. In both *calc*2 and *calc*4, the additional test cases introduced no redundancy into the test suite; in all other cases, redundant test cases were found.

Although the programs tested are small, our experiments show that a considerable reduction can be made in the size of the test suite. The reduction for our examples ranged from 19% to 60% when actually developing the program. The reduction ranged from 20% to 55% for implementation changes and from 0% to 40% when making program enhancements. For larger programs, we expect our technique to scale up favorably. For a program with a few lines of code, each test case in the test suite may satisfy a small number of test case requirements. However, for a larger program, each test case may satisfy many test case requirements. Thus, test cases developed to satisfy specific test case requirements may cause redundancy. For example, for unit testing, test cases are developed for individual procedures. However, when the procedures are integrated, many of the test cases may be redundant and can be eliminated.

7. CONCLUSION

We have presented a technique that helps manage a test suite by identifying redundant and obsolete test cases. The identification of unnecessary test cases can be used to reduce the size of the test suite by eliminating both obsolete and redundant test cases. This identification can also be used to select test cases to run by eliminating the obsolete and redundant test cases from consideration. The technique is not dependent on any particular test selection criterion and can be used as long as the association between requirements and test cases can be made. The technique can either be integrated into a testing methodology or be used as a stand-alone tool. Thus, as a program is being developed and tested, the methodology could be used to identify unnecessary test cases. Experimentation shows that even for small programs, a significant reduction in the test suite may be realized.

REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass. 1986.

2. BEIZER, B. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.
3. FRANKL, P. G., AND WEYUKER, E. J. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. SE-14*, 10 (Oct. 1988), 1483–1498.
4. GAREY, M. R., AND JOHNSON, D. S. *Computers and intractability, A Guide to the Theory of NP-Completeness*, V. Klee, Ed. Freeman, New York, 1979.
5. GUPTA, R. A reconfigurable LIW architecture and its compiler. Tech. Rep. 87-3. Dept. Computer Science, Univ. Pittsburgh, Pittsburgh, Pa., 1987.
6. GUPTA, R., AND SOFFA, M. L. Compile-time techniques for improving scalar access performance in parallel memories. *IEEE Trans. Parallel and Distruibuted Systems 2*, 2 (Apr. 1991), 138–148.
7. HARROLD, M. J. An approach to incremental testing. Tech. Rep. 89-1. Ph.D dissertation. Dept. Computer Science, Univ. Pittsburgh, Pittsburgh, Pa., 1989.
8. HARROLD, M. J., AND KOLTE, P. Combat: A compiler-based data flow testing system. In *Proceedings of Pacific Northwest Quality Assurance* (Oct.). Lawrence and Craig, 1992, 311–323.
9. HARROLD, M. J., AND SOFFA, M. L. Computation of interprocedural definition and use dependencies. In *Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages*, (New Orleans, La., Mar.). IEEE, 1990, 297–306.
10. HARROLD, M. J., AND SOFFA, M. L. An incremental data flow testing tool. In *Proceedings of the 6th International Conference on Testing Computer Software* (Washington, D.C., May). USPDI, 1989.
11. HARROLD, M. J., AND SOFFA, M. L. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance 1988* (Oct.). IEEE, New York, 1988, 362-367.
12. LASKI, J. W., AND KOREL, B. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. SE-9*, 3 (May 1983), 347–354.
13. LEUNG, H. K., AND WHITE, L A cost model to compare regression test strategies. In *Proceedings of Conference on Software Maintenace 1991* (Oct.). IEEE, New York, 1991, 201–208.
14. LEUNG, H. K. L., AND WHITE, L. A study of regression testing. In *Proceedings of the 6th International Conference on Testing Computer Software* (Washington, D.C., May). USPDI, 1989.
15. NTAFOS, S. C. An evaluation of required element testing strategies. In *Proceedings of the 7th International Conference on Software Engineering* (Mar.). IEEE, New York, 1984, 250–256.
16. OSTRAND, T. J., AND WEYUKER, E. J. Using data flow analysis for regression testing. In *Proceedings of the 6th Annual Pacific Northwest Software Quality Conference* (Sept.). IEEE, New York, 1988, 58–71.
17. RAPPS, S., AND WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr. 1985), 367–375.
18. TAHA, A. M., THEBUT, S. M., AND LIU, S. S. An approach to software fault localization and revalidation-based on incremental data flow analysis. In *Proceedings of COMPSAC 89* (Sept.). IEEE, New York, 1989, 527–534.

Received September 1992; revised March 1993; accepted April 1993