Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage

James A. Jones and Mary Jean Harrold, *Member*, *IEEE Computer Society*

Abstract—Software testing is particularly expensive for developers of high-assurance software, such as software that is produced for commercial airborne systems. One reason for this expense is the Federal Aviation Administration's requirement that test suites be modified condition/decision coverage (MC/DC) adequate. Despite its cost, there is evidence that MC/DC is an effective verification technique and can help to uncover safety faults. As the software is modified and new test cases are added to the test suite, the test suite grows and the cost of regression testing increases. To address the test-suite size problem, researchers have investigated the use of test-suite reduction algorithms, which identify a reduced test suite that provides the same coverage of the software according to some criterion as the original test suite, and test-suite prioritization algorithms, which identify an ordering of the test cases in the test suite according to some criteria or goals. Existing test-suite reduction and prioritization techniques, however, may not be effective in reducing or prioritizing MC/DC-adequate test suites because they do not consider the complexity of the criterion. This paper presents new algorithms for test-suite reduction and prioritization that can be tailored effectively for use with MC/DC. The paper also presents the results of empirical studies of these algorithms.

Index Terms—Test-suite reduction, test-suite prioritization, modified condition/decision coverage, testing, critical software.

1 Introduction

To facilitate the testing of evolving software, a test suite is typically developed for the initial version of the software and reused to test each subsequent version of the software. As new test cases are added to the test suite to test new or changed requirements or to maintain test-suite adequacy, the size of the test suite grows and the cost of running it on the modified software (i.e., *regression testing*) increases.

Regression testing is particularly expensive for developers of high-assurance software, such as software that is produced for commercial airborne systems. One reason for this expense is the extensive verification of the software required for Federal Aviation Administration approval. Such approval includes the requirement that a test suite be adequate with respect to modified condition/decision coverage (MC/DC). For example, one of our industrial partners reports that, for one of its products of about 20,000 lines of code, the MC/DC-adequate test suite requires seven weeks to run. Despite its cost, there is evidence that MC/DC is an effective verification technique. For example, a recent empirical study performed during the real testing of the attitude-control software for the HETE-2 (High Energy Transient Explorer) found that the test cases generated to satisfy the MC/DC coverage requirement detected important errors not detectable by functional testing.[1]

1. Private communication with Nancy Leveson of MIT.

• *The authors are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.*
E-mail: {jjones, harrold}@cc.gatech.edu.

Manuscript received 22 Mar. 2002; revised 25 July 2002; accepted 10 Sept. 2002.
Recommended for acceptance by G. Canfora and A.A. Andrews.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117301.

Researchers have investigated two approaches for addressing the test-suite size problem that maintain the same coverage as the original test suite:[2] test-suite reduction and test-suite prioritization. *Test-suite reduction* (also known as test-suite minimization) algorithms (e.g., [1], [7], [9], [11], [14], [15]) identify a reduced test suite that provides the same coverage of the software as the original test suite. *Test-suite prioritization* algorithms (e.g., [5], [4], [12]) identify an ordering of the test suite according to some criteria.

Existing test-suite reduction and prioritization techniques consider test-case coverage criteria (e.g., statements, decisions, definition-use associations, or specification items), other criteria such as risk or fault-detection effectiveness, or combinations of these criteria. As we discuss in Section 4, for test-suite reduction and prioritization based on coverage, there are important differences between coverage criteria such as statement and MC/DC. Thus, existing techniques may not be effective for use in reducing or prioritizing MC/DC-adequate test suites because they do not consider the complexities of the criterion. However, because of the enormous testing expense that users of MC/DC can incur, effective test-suite reduction and prioritization techniques that can be applied to MC/DC-adequate test suites could provide significant savings to developers who use this powerful test-coverage criterion.

This paper presents two new algorithms for test-suite reduction and one new algorithm for test-suite prioritization that can incorporate aspects of MC/DC effectively. Unlike existing algorithms, when making decisions about reducing or ordering a test suite, our algorithms consider the complexities of MC/DC. This paper also presents

2. Another related approach that addresses the test-suite size problem is test selection (e.g., [2], [10]), which selects a subset of the test suite that will execute code or entity changes; this test suite, however, may not provide the same coverage as the original test suite.

empirical studies that evaluate the effectiveness of our test-suite reduction algorithms, compare the benefits of each, and evaluate the performance of our test-suite prioritization algorithm.

This paper addresses the issues surrounding test-suite reduction and prioritization for MC/DC. The main contributions of this paper are:

- An analysis of the problems that arise when attempting to apply existing test-suite reduction and prioritization algorithms for use with MC/DC. This analysis shows that new techniques are necessary for effective test-suite reduction and test-suite prioritization for MC/DC.
- A description of a general approach to designing algorithms for test-suite reduction and prioritization along with a detailed description of these algorithms—two for reduction and one for prioritization. These algorithms use MC/DC coverage information to characterize the strength of test cases.
- A set of empirical studies performed on two real C subjects that compare the relative performance and effectiveness of the two reduction algorithms and present a performance evaluation of the prioritization algorithm. These studies show that our test-suite reduction techniques can be effective in reducing test suites while providing acceptable performance. The two techniques provide a trade off between performance and reduction: For our subject programs, versions, and test suites, one technique runs in linear time in the size of the original test suite, but provides slightly larger reduced test suites, whereas the other runs in quadratic time in the size of the original test suite and provides slightly smaller reduced test suites. The study also evaluates the time required to prioritize a test suite.

In the next section, we present background on MC/DC. Section 3 presents the test-suite reduction and the test-suite prioritization problems, discusses attributes that must be considered in designing techniques to address these problems, and motivates the need for new algorithms for MC/DC. Section 4 presents three possible algorithms—two for test-suite reduction and one for test-suite prioritization—tailored for MC/DC. In Section 5, we present empirical results that evaluate each algorithm individually and that compare the algorithms to each other. Finally, Section 6 presents conclusions and discusses some future work.

2 Modified Condition/Decision Coverage

MC/DC is a stricter form of decision (or branch) coverage. For decision coverage, each decision statement must evaluate to `true` on some execution of the program and must evaluate to `false` on some execution of the program.[3] MC/DC, however, requires execution coverage at the condition level. A *condition* is a Boolean-valued expression that cannot be factored into simpler Boolean expressions.

3. If no inputs cause these `true` or `false` evaluations for the predicate, the decision is *infeasible* and cannot be covered by any test case.

Program P

S1: if (A or B) and C

⋮

S2: if (not (D and E)) and F

⋮

end P

(a)

S1: if (A or B) and C

A(T)	A(F)
TFT	FFT

B(T)	B(F)
FTT	FFT

C(T)	C(F)
FTT	FTF
TFT	TFF
TTT	TTF

S2: if (not(D and E)) and F

D(T)	D(F)
FTT	TTT

E(T)	E(F)
TFT	TTT

F(T)	F(F)
FTT	FTF
TFT	TFF
FFT	FFF

(b)

	S1	S2
T1	TFT	TFT
T2	FTF	FTF
T3	FFT	FTT
T4	FTT	TFF
T5	TFF	TTT
T6	FTT	TTT

(c)

A(T)	A(F)
T1	T3

B(T)	B(F)
T4, T6	T3

C(T)	C(F)
T4, T6	T2
T1	T5
--	--

D(T)	D(F)
T3	T5, T6

E(T)	E(F)
T1	T5, T6

F(T)	F(F)
T3	T2
T1	T4
--	--

(d)

Fig. 1. (a) Partial program P; (b) its truth vectors, organized by condition, truth value, and MC/DC pair; (c) test cases and their coverage; (d) subfigure (b) with the truth vectors replaced by the test cases that cover them.

For example, the partial program P in Fig. 1a contains a decision statement, S1, that has three conditions: A, B, and C.

MC/DC requires that each condition in a decision be shown by execution to independently affect the outcome of the decision [3]. Each possible evaluation of that decision produces a *truth vector*—a vector of the Boolean values resulting from the evaluation of the conditions in that decision. For example, "TTF" in Figs. 1 and 2 for statement S1 is a truth vector in which condition A

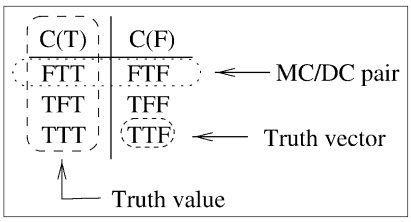

Fig. 2. Condition C (Fig. 1) and its constituent parts.

evaluates to `true`, B evaluates to `true`, and C evaluates to `false`.

An *MC/DC pair* is a pair of truth vectors, each of which causes a different result for the decision statement, but that differ only by the value of one condition. For example, the truth vectors "FTT" and "FTF" for condition C in Fig. 2 differ only in the evaluation of C and cause different evaluations of the decision. Thus, these two truth vectors comprise an MC/DC pair for condition C. The truth vector "FTF" is called "FTT"'s *mate truth vector* and vice versa.

A *truth value* for a condition *cond* is the set of truth vectors belonging to the MC/DC pairs for *cond* that cause *cond*'s decision to evaluate to a particular result. Fig. 2 illustrates C's truth values: {FTT, TFT, TTT} for C(T) and {FTF, TFF, TTF} for C(F). Note that, although the truth vector "FFF" causes C's decision to evaluate to `false`, it is not an element of the truth value for C(F) because it is not part of an MC/DC pair for C.

In some cases, there can be more than one MC/DC pair for a condition. Fig. 1b shows, for example, that there is only one MC/DC pair for conditions A, B, D, and E, but there are three MC/DC pairs for conditions C and F, each of which can be used to show the independence of the condition. For condition C, for example, there are three MC/DC pairs; we denote them as C_i, where i is the row number in the table for C. Because MC/DC requires that MC/DC pairs be covered, the coverage of truth vectors "FTT" and "TFF," although satisfying $\mathrm{C}_1(\mathrm{T})$ and $\mathrm{C}_2(\mathrm{F})$, respectively, for condition C, does not satisfy the MC/DC criterion for C.

3 Test-Suite Reduction and Prioritization

The *optimal test-suite reduction problem* can be stated as:[4]

Given: Test suite T, a set of test-case requirements, $r_1, r_2, \ldots, r_n$, that must be satisfied to provide the desired test coverage of the program.

Problem: Find $T' \subset T$ such that T' satisfies all r_is and $(\forall T'')$ $(T'' \subset T)$ (T'' satisfies all r_is) $[|T'| \leq |T''|]$.

Given subsets of T, $T_1, T_2, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to test r_i, a test suite that satisfies all r_is must contain at least one test case from each T_i. Such a set is called a hitting set of the T_i. Maximum reduction is achieved with the minimum cardinality hitting set of the T_is. Because the problem of finding the minimum cardinality hitting set is intractable [6], test-suite reduction techniques must be heuristic.

The *test-suite prioritization problem* can be stated as:[5]

Given: Test suite T, a set of permutations of T, PT, a function f from PT to the real numbers.

Problem: Find $pt \in PT$ such that $(\forall pt')$ $(pt' \in PT)$

$$[f(pt) \geq f(pt')].$$

PT represents the set of possible orderings (prioritizations) of T, and f is a function that, when applied to an ordering, yields an evaluation of that ordering. Depending on the choice of f, the test-case prioritization problem may be intractable or undecidable. For example, given a function f that quantifies the rate at which a test suite achieves statement coverage, efficiently determining the test suite that maximizes f would provide an efficient solution to the knapsack problem [12].[6] Similarly, given a function f that quantifies the rate at which a test suite detects faults, a precise solution to the prioritization problem would provide a solution to the halting problem [12]. In such cases, prioritization techniques must be heuristic.

Test-suite reduction algorithms input a test suite T and output a new test suite, whereas test-suite prioritization algorithms input a test suite T and output a sequence of test cases. For convenience, in the following discussion, we refer to both outputs as T'.

Test-suite reduction and prioritization algorithms share a number of common attributes. First, in constructing T', the algorithms evaluate each test case for its *contribution* or *goodness* based on some characteristics of the program under test $\mathcal{P}$, some characteristics of the test cases in T, and the goal of T'. Although a variety of characteristics have been considered, we can classify them as either types of coverage or types of cost. *Types of coverage* include $\mathcal{P}$'s requirements, code-based criteria such as statement, branch, and data-flow, and risks, age, or error-proneness of components of $\mathcal{P}$. *Types of cost* include execution time for $\mathcal{P}$ with a test case, set-up time for preparing to run a test case, and any financial cost associated with running a test case (especially in the case of simulations). The algorithms evaluate test cases in T based on some combination of these characteristics. For example, Harrold et al. [7] and Rothermel et al. [11] use various types of coverage for test-suite reduction. For each program entity E, this technique uses the number of test cases that cover E when making decisions about which test cases to include in T'. For another example, Elbaum et al. [5] and Rothermel et al. [12] use various types of coverage, along with error-proneness of modules, to select an ordering of test cases for T'.

Second, test-suite reduction and prioritization algorithms may vary in the frequency with which they recompute the contribution of a test case. One approach computes this contribution once and uses it to select test cases for inclusion in or exclusion from T'. Rothermel et al. [12] refer to this frequency as *total* and use it to compute total-based prioritization. Another approach recomputes the contribution after test cases are included in or excluded from T' based on the additional contribution of the test case to T'.

4. Variation of the problem presented by Harrold et al. [7, p. 272].
5. Problem and argument presented by Rothermel et al. [12, p. 3].

6. Informally, the knapsack problem is the problem of, given a set U whose elements each have a cost and a value and given a size constraint and a value goal, finding a subset U' of U such that U' meets the given size constraint and the given value goal. For a more formal treatment, see [6].

Rothermel et al. [12] refer to this frequency as *additional* and use it to compute additional-based prioritization. In discussing our algorithms, we use Rothermel et al.'s terminology to refer to techniques for computing a test case's contribution: Total refers to an a priori evaluation in which test cases are evaluated based only on the characteristics of the test case, whereas additional refers to an evaluation that considers the current state of the reduction or prioritization as well as characteristics of the test case. In the case of additional evaluation, the frequency with which test cases are re-evaluated may vary, and re-evaluation may occur after n iterations.

Third, an algorithm can use a test case's contribution for determining the next test case to add to T' (for both reduction and prioritization) or, alternatively, for determining the next test case to remove from T (for reduction). We refer to the former as a *build-up* technique and to the latter as a *break-down* technique. In a build-up technique, the algorithm begins with an empty T' and adds test cases to it, whereas, in a break-down technique, the algorithm begins with T and removes test cases from it to get T'. One important advantage of a break-down approach for test-suite reduction is that the algorithm can be stopped at any time during the reduction process, and the remaining test suite (i.e., $T - \{\text{removed test cases}\}$) provides coverage of the test-case requirements. This coverage can be important if manual intervention is required in making decisions for test-case removal. In contrast, a build-up approach can guarantee coverage only when the algorithm terminates.

Combinations of these attributes—computation of test-case contribution, frequency of (re)evaluation of test-case contribution, and method of constructing T'—can produce a number of different algorithms. In the next section, we describe several such algorithms, which we have developed, that consider the complexities of MC/DC.

4 Test-Suite Reduction and Prioritization for MC/DC

In this section, we first discuss the problems with using existing test-suite reduction and prioritization algorithms for MC/DC. Next, we present three new algorithms—two for test-suite reduction and one for test-suite prioritization—that consider the complexities of MC/DC.

4.1 Using Existing Algorithms for MC/DC

For test-suite reduction and prioritization, there are two important differences between coverage criteria, such as statement, and MC/DC. First, coverage criteria, such as statement, require that every entity (e.g., statement) be covered by the test suite. For these criteria, a test case either covers an entity or does not cover the entity; we call such criteria *single-entity* criteria. In contrast, MC/DC requires that every condition be covered by the execution of an MC/DC pair. For MC/DC, a test case may cover an MC/DC pair, but, more often, a test case contributes to the coverage of a condition by covering *only one* of its truth vectors; we call such criteria *multiple-entity* criteria. Second, coverage criteria, such as statement, have only one way to be covered—the entity (e.g., statement) must be executed by one of the test cases in the test suite. For MC/DC, there may be several MC/DC pairs associated with a condition, and executing any of them provides coverage of the condition. For example, in partial program P of Fig. 1a, condition F has three possible MC/DC pairs and coverage of only one of them is necessary to satisfy MC/DC for F.

Because of these differences, existing test-suite reduction and prioritization techniques, which were developed for single-entity criteria, are not sufficient when applied to multiple-entity criteria such as MC/DC. Existing test-suite reduction and prioritization techniques require an association between entities to be covered and test cases in the test suite. In attempting to use the existing techniques for MC/DC, the major problem is the identification of the entities to be covered. To illustrate the problem, consider two possible approaches for test-suite reduction; approaches for test-suite prioritization have similar problems.

Under the first approach, for test-suite reduction, the r_is are the MC/DC pairs. The r_is would be, for example (Fig. 1), MC/DC pair ("TFT," "FFT") for A and ("FTT," "FTF"), ("TFT," "TFF"), and ("TTT," "TTF") for C. Because, in most cases, two test cases are required to cover an MC/DC pair, we may not be able to get a relationship between test cases and MC/DC pairs. In fact, for the MC/DC pairs in program P, no MC/DC pair is covered by a single test case. Thus, we cannot associate individual test cases with MC/DC pairs. Because we cannot get this association of MC/DC pairs and test cases, we cannot use MC/DC pairs as the coverable entities for the reduction.

Under the second approach, for test-suite reduction, the r_is are the truth values for conditions. For example, the truth value A(T) would be satisfied with execution of the truth vector "TFT," and the truth value C(T) would be satisfied with the execution of any one of the truth vectors "FTT," "TFT," and "TTT." With this approach, we can associate individual test cases with truth values, and MC/DC is like the class of single-entity criteria in that a test case covers (or does not cover) each entity. This approach, however, is not sufficient for test-suite reduction because MC/DC requires that an MC/DC pair of truth vectors be covered. For example, C(T) may be satisfied by executing truth vector "FTT," and C(F) may be satisfied by executing truth vector "TFF." In this case, both true and false entities are covered for condition C. However, MC/DC is not satisfied for C because the truth vectors do not constitute an MC/DC pair.

A naive approach for providing a test-suite reduction algorithm for MC/DC is to associate, with each MC/DC pair, the *pairs* of test cases that cover the pair. Given this association, we can apply Harrold et al.'s (HGS) algorithm for reduction of the test suite [7].

Table 1 shows the association between MC/DC pairs and test-case pairs for our example program P (Fig. 1a). The HGS algorithm would be applied to these test-case pairs instead of to the individual test cases. This application of the HGS algorithm would not take advantage of the intersection of test-case pairs in the reduction process. The resulting T would be {T1, T3, T4, T5, T6}.

Although this approach can provide some reduction in the test suite, the algorithm does not consider individual test cases and the contribution they make to MC/DC. We

TABLE 1
MC/DC Pairs and Associated Test-Case Pairs for Program P of Fig. 1

Condition	Test-Case Pair
A	{(T1, T3)}
B	{(T3, T4), (T3, T6)}
C	{(T2, T4), (T2, T6), (T1, T5)}
D	{(T3, T5), (T3, T6)}
E	{(T1, T5), (T1, T6)}
F	{(T1, T4), (T2, T3)}

believe, and our case study suggests, that we can achieve more reduction in test-suite size using an algorithm that considers the partial coverage of the MC/DC pairs by individual test cases. The next section presents this new algorithm and the following section presents a test-suite prioritization algorithm that considers individual test cases and the contribution they make to MC/DC.

4.2 New Test-Suite Reduction Algorithm (Break-Down)

Our first test-suite reduction algorithm bases its contribution computation on MC/DC pairs. The algorithm utilizes an additional approach that recomputes the contribution of test cases after each test case is selected and uses a break-down approach that removes the weakest test case from the test suite.

To describe test-suite reduction, we define two important terms: test-case redundancy and test-case essentiality. A test case t is *redundant* with respect to a set of test cases T if the set of test-case requirements covered by t is a subset of the set of test-case requirements covered by T. To eliminate some test case t from a test suite T and retain the coverage of T, t must be redundant with the remaining test cases in T (i.e., $T - \{t\}$). A test case t is *essential* with respect to a set of test cases T if the set of test-case requirements covered by t is not a subset of the set of test-case requirements covered by T and, thus, t is not redundant with respect to T. A test case that uniquely covers any test-case requirement must be present in the reduced test suite to retain coverage of that requirement. Every test case t in a test suite T can be identified as either essential or redundant with respect to $T - \{t\}$.

Specifically, for MC/DC, a test case is essential if it uniquely covers a truth value of a condition after uncovered MC/DC pairs have been removed. The uncovered MC/DC pairs are removed because they do not contribute to the coverage of the condition. Fig. 3 demonstrates the process of discovering essential test cases: First, we view the test-case coverage of the MC/DC pairs, as shown in Fig. 3a; then, we remove any MC/DC pairs that are uncovered, as shown in Fig. 3b with the removal of the second MC/DC pair; finally, we identify the test cases that uniquely cover a truth value, as shown in Fig. 3c with the dashed box around the pair of occurrences of T1. Test case T1 is identified as essential because it is the only test case that covers the X(T) truth value that can aid in the coverage for the condition. This process is applied to all covered conditions (i.e., conditions with a covered MC/DC pair). Conditions that are uncovered by the original test suite are handled differently; we explain this in the Other considerations section of Section 4.2.

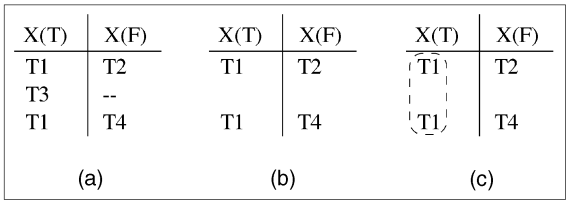

Fig. 3. (a) Example condition shown with test-case coverage, (b) removal of uncovered MC/DC pairs, (c) identification of essential test cases.

Our algorithm first identifies essential test cases; the remaining test cases in the test suite are redundant with respect to some other subset of test cases in the suite. The algorithm then iteratively identifies the *weakest* test case—the test case that contributes the least to coverage of the test-case requirements—of the test cases that have not been marked as essential. The algorithm then eliminates that test case and identifies new essential test cases. When all test-case requirements covered by the original test suite are covered by the set of essential test cases, the algorithm halts, and the essential test cases are returned as the reduced test suite.

Our algorithm `ReduceSuite`, shown in Fig. 4, can be applied to both single-entity and multiple-entity criteria. `ReduceSuite` inputs 1) $allTests$—the test cases in the test suite, annotated with the conditions that they cover and 2) $allConditions$—the conditions in the program under test $\mathcal{P}$ annotated with the test cases that cover them. The algorithm outputs the test cases comprising the reduced test suite. Step 1 is executed once at the beginning of the algorithm, and then Steps 2, 3, and 4 are executed iteratively until $allConditions$ is empty.

Step 1: Eliminate uncovered MC/DC pairs. In Step 1 (lines 2-4), any MC/DC pairs that are not covered by $allTests$ are removed from $allConditions$. These MC/DC pairs cannot contribute to the MC/DC coverage. For example, the MC/DC pairs $(\mathrm{C}_3(\mathrm{T}), \mathrm{C}_3(\mathrm{F}))$ and $(\mathrm{F}_3(\mathrm{T}), \mathrm{F}_3(\mathrm{F}))$[7] in conditions C and F, respectively, in Fig. 1 are removed. Fig. 5, which is modified from Fig. 1d, illustrates the state of the example after Step 1 of `ReduceSuite`.

Step 2: Identify essential test cases. In the second step (lines 6-19), `ReduceSuite` identifies the test cases that are essential with respect to $allTests$. The algorithm examines each condition, c, in $allConditions$ to identify any test case that uniquely covers either c's `true` truth value or c's `false` truth value (lines 6-9). When the algorithm finds such a test case t, it adds t to $essentialTests$ and removes t from $allTests$ (lines 10-11). Note that $essentialTests$ is kept as a convenience rather than a necessity. The essential test cases could also be marked essential in $allTests$, but, for ease and speed, a separate set is used (however, this is not a build-up algorithm). The algorithm then examines each truth vector v that t covers to see whether any of the test cases that cover v's mate truth vector are essential (line 13); this indicates a covered MC/DC pair and, thus, a covered condition. When the algorithm finds such a case, it removes the condition to which v belongs from $allConditions$ (line 14).

7. Recall from Section 2 that C_i is the ith MC/DC pair in condition C.

```
algorithm   ReduceSuite(allTests, allConditions)
input       allTests: set of test cases for P
            allConditions: set of conditions for P
output      essentialTests: reduced test suite
begin ReduceSuite
1.  essentialTests = φ
2.  for each condition, c, in allConditions do                    /*Step 1*/
3.    remove all uncovered MC/DC pairs from c
4.  endfor
5.  do
6.    for each condition, c, in allConditions do                  /*Step 2*/
7.      for x in {true,false} do
8.        if the (# of test cases covering all of the x truth vectors
              in c) = 1 then
9.          t = the test covering x truth vectors in c
10.         essentialTests = essentialTests ∪ {t}
11.         allTests = allTests − {t}
12.         for each truth vector, v, covered by t do
13.           if (the set of test cases covering v's mate truth vector
                in its MC/DC pair) ∩ essentialTests ≠ φ then
14.             allConditions = allConditions − {condition
                  to which v belongs}
15.           endif
16.         endfor
17.       endif
18.     endfor
19.   endfor
20.   for each test, t, in allTests                               /*Step 3*/
21.     t.contribution = 0
22.   endfor
23.   for each condition, c, in allConditions do
24.     for x in {true,false} do
25.       indivContrib = indivContrib + 1.0/(# of test cases
            covering all of the x truth vectors in c)
26.       for each x truth vector, v, in c do
27.         for each test, t, covering v do
28.           t.contribution = t.contribution + indivContrib
29.         endfor
30.       endfor
31.     endfor
32.   endfor
33.   t = test in allTests with lowest contribution               /*Step 4*/
34.   allTests = allTests − {t}
35.   for each truth vector, v, that t covers do
36.     v.coveringTests = v.coveringTests − {t}
37.     if (the MC/DC pair, p, to which v belongs becomes
          uncovered) then
38.       remove p from its condition
39.     endif
40.   endfor
41. while (allConditions ≠ φ)
42. return essentialTests
end ReduceSuite
```

Fig. 4. `ReduceSuite`: MC/DC adequate test suite reduction.

A(T)	A(F)
T1	T3

D(T)	D(F)
T3	T5, T6

B(T)	B(F)
T4, T6	T3

E(T)	E(F)
T1	T5, T6

C(T)	C(F)
T4, T6	T2
T1	T5

F(T)	F(F)
T3	T2
T1	T4

allTests = {T1,T2,T3,T4,T5,T6}

allConditions = {A,B,C,D,E,F}

essentialTests = { }

Fig. 5. Example after Step 1 of `ReduceSuite`.

A(T)	~~A(F)~~
T1	T3

D(T)	D(F)
T3	T5, T6

B(T)	B(F)
T4, T6	T3

E(T)	E(F)
T1	T5, T6

C(T)	C(F)
T4, T6	T2
T1	T5

F(T)	F(F)
T3	T2
T1	T4

allTests = {T2,T4,T5,T6}

allConditions = {B,C,D,E,F}

essentialTests = {T1,T3}

Fig. 6. Example after first iteration of Step 2 of `ReduceSuite`.

In the example of Fig. 1, because T1 is the only test case that covers A(T), `ReduceSuite` identifies it as an essential test case, removes it from $allTests$, and adds it to $essentialTests$. Likewise, the algorithm identifies T3 as essential, removes it from $allTests$, and adds it to $essentialTests$. Next, the algorithm identifies condition A as covered by $essentialTests$ and removes it from $allConditions$. Fig. 6 shows the state of the example after the first iteration of Step 2. The dotted boxes denote the essential test cases and the slashed condition denotes a condition covered by $essentialTests$.

Step 3: Assign test-case contributions. After the algorithm identifies all essential test cases and removes them from $allTests$ (Step 2), each remaining test case t in $allTests$ is redundant with $essentialTests \cup allTests - \{t\}$. Thus, the algorithm can remove any test case from $allTests$ and retain coverage of $\mathcal{P}$. In Step 3 (lines 20-32), the algorithm attempts to find the test case that contributes least to the coverage in $allTests$. The algorithm does this by first initializing the $contribution$ of each test case to zero (lines 20-22). Then, for each condition c, the algorithm increments the contribution of each test case that covers each of c's `true` and `false` truth values tv by $(1.0/\text{the number test cases covering } tv)$ (lines 23-32).

In the example, the contributions given to the test cases in $allTests$ are shown in Table 2. To illustrate, the contribution of $1.\overline{3}$ for T4 is achieved by summing the individual contribution scores for its coverage of B(T), C(T), and F(F) (i.e., 0.5, $0.\overline{3}$, and 0.5, respectively).

Step 4: Discard weakest test case. In Step 4 (33-40), the algorithm eliminates the weakest test case—the test case

TABLE 2
Test Cases for Example and Associated Contribution Weights Assigned during Step 3

Test Case	Contribution Weight
T2	1.0
T4	$1.\overline{3}$
T5	1.5
T6	$1.8\overline{3}$

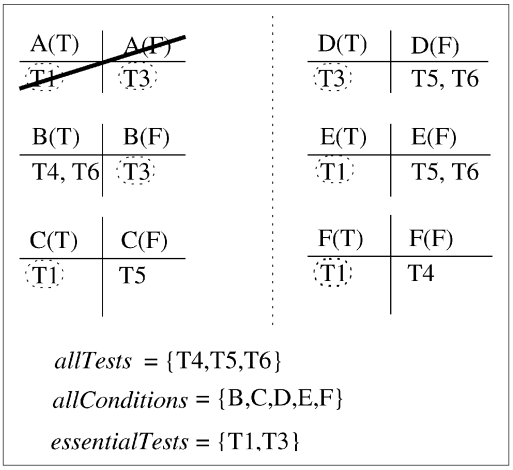

Fig. 7. Example after first iteration of Step 4 of `ReduceSuite`.

that contributes least to the coverage of $\mathcal{P}$, according to the heuristic given in Step 3. The test case with the lowest contribution in $allTests$, t, is removed from $allTests$ (line 34). For each truth vector v that was covered by t, the algorithm removes t from the set of test cases covering v (lines 35-40). If the MC/DC pair to which v belongs becomes uncovered as a result of discarding t, then the algorithm removes the MC/DC pair from its condition (37-39). Fig. 7 shows the state of the example after the test case with the lowest contribution T2 is discarded, and this step has completed.

Steps 2, 3, and 4 are repeated until $allConditions$ is empty and, thus, all conditions in $\mathcal{P}$ are covered by a set of essential test cases. In the next iteration of the algorithm, T4 and T5 are identified as essential. They are thus removed from $allTests$ and added to $essentialTests$. At this point, conditions B, C, D, E, and F are covered by the test cases in $essentialTests$ and are thus removed from $allConditions$. T6 is discarded because it is the only test in $allTests$ and, consequently, has the lowest contribution. The loop halts because $allConditions$ is empty and $essentialTests$ is returned as the reduced test suite. Fig. 8 shows the state of the example after the next iteration of Steps 2, 3, and 4.

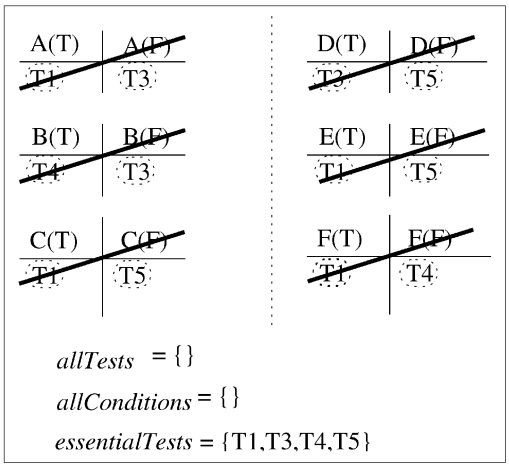

Fig. 8. Example after second iteration of Steps 2, 3, and 4 of `ReduceSuite`.

```
algorithm   PrioritizeSuite(allTests, allEntities)
input       allTests: set of test cases for P
            allEntities: set of entities for P
output      orderedTestSuite: prioritized test suite
begin PrioritizeSuite
1.  mark all entities in allEntities uncovered                    /* Step 1 */
2.  orderedTestSuite = empty array of size allTests
3.  orderedTestSuite[0] = {t ∈ allTests|∀t' ∈ allTests, entity
        coverage of t' ≤ entity coverage of t}
4.  allTests = allTests − {t}
5.  mark all entities in allEntities, covered, that are covered by t
6.  i = 1
7.  while allTests ≠ φ do                                         /* Step 2 */
8.    for each test, t, in allTests do
9.      t.contribution = the sum of the # of MC/DC pairs
            completed and the number of entries, exits, cases,
            and truth vectors for uncovered conditions covered
10.   endfor
11.   if the highest contribution = 0 then
        /* if all conditions, entries, exits, and cases are covered,
          orderedTestSuite can be used as a reduced test suite */
12.     mark all entities in allEntities uncovered
13.     nextTest = {t ∈ allTests|∀t' ∈ allTests, entity
            coverage of t' ≤ entity coverage of t}
14.   else
15.     nextTest = {t ∈ allTests|∀t' ∈ allTests,
            t'.contribution ≤ t.contribution}
16.   endif
17.   orderedTestSuite[i] = nextTest
18.   allTests = allTests − {nextTest}
19.   mark all entities in allEntities, covered, that are covered by
            nextTest
20.   i = i + 1
21. endwhile
22. return orderedTestSuite
end PrioritizeSuite
```

Fig. 9. `PrioritizeSuite`: MC/DC-based test-suite prioritization.

Other considerations. To simplify our discussion, we presented our algorithm `ReduceSuite` for test suites that are MC/DC-adequate. However, with minor modification, the algorithm can reduce test suites that are not MC/DC-adequate. In this case, there are uncovered or partially covered MC/DC pairs. For uncovered or partially covered conditions, in Step 1 the algorithm eliminates all MC/DC pairs that have no coverage: Neither truth vector in the MC/DC pair is covered by any test case. If all MC/DC pairs are removed from a condition, then the condition is not covered at all, and it is removed from $allConditions$. For partially covered conditions, we define essentiality slightly differently. A test case is considered to be essential if it uniquely covers a truth *vector* of a partially covered condition. Fig. 10 shows test cases T1, T2, and T3—all of which are essential. We treat partially covered conditions in this way to ensure that if, at some later time, new test cases are added to the reduced test suite, that test suite will preserve the original test suite's potential for coverage. For example, if partially covered conditions were treated in the same manner as covered conditions in Fig. 10, we would need only one test case in {T1, T3} to be in the reduced test suite. Suppose that T1, but not T3, were chosen to be in the reduced test suite. If a test case were added to the test suite that covered X(F) in the third MC/DC pair, coverage would not be achieved for this condition.

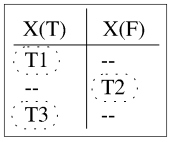

Fig. 10. Example partially covered condition with essential test cases T1, T2, and T3.

The MC/DC criterion also specifies that all entry and exit points, and all case statements in the program be exercised. A simple extension to `ReduceSuite` handles entry and exit points and case statements by stating that essentiality is achieved when one test case covers each of these entities.

4.3 New Test-Suite Prioritization Algorithm

Like our break-down reduction algorithm, our test-suite prioritization algorithm, shown in Fig. 9, bases its contribution computation on MC/DC pairs and utilizes an additional approach that recomputes the contribution of test cases after each test case is selected. However, instead of the test-case evaluation being based on the uniqueness of program-entity coverage, this algorithm uses a simpler evaluation based on additional MC/DC pairs covered. In contrast to our break-down test-suite reduction, the test-suite prioritization algorithm uses a build-up approach that adds the "strongest" test case to the new test suite.

Step 1: Initialization. In Step 1, `PrioritizeSuite` first selects the test case in the test suite that has the highest entity coverage (sum of the test vectors, procedure entries, procedure exits, and cases covered) (line 3). This test case is placed into an ordered list $orderedTestSuite$ as the first test case and removed from the set containing the original test suite $allTests$. The algorithm augments each program entity with a flag that denotes its coverage by a test case (lines 1, 5, 12, 19). For example, for the conditions and coverage shown in Fig. 1, Table 3 shows the entity coverage of each test case, with T1, T3, and T6 tied for the most entities covered. Test case T1 is chosen as the first test case to be placed into $orderedTestSuite$. The coverage of the truth vectors for which T1 covers is shown with the dotted boxes around the instances of T1 in Fig. 11.

Step 2: Additional prioritization. The algorithm then iterates over the statements in Step 2 until $allTests$ is empty. In each iteration, all test cases in $allTests$ are assigned a contribution value (lines 8-10). The contribution, or goodness, for test case t is evaluated based on the number of MC/DC pairs, entries, exits, and case statements whose coverage is *completed* by t—that is, those coverage entities that are uncovered by the test cases in $orderedTestSuite$, but are covered by $\{t\} \cup \{\text{test cases in } orderedTestSuite\}$. The

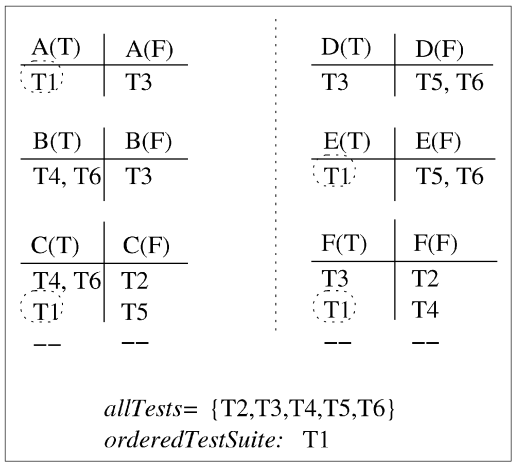

Fig. 11. Example after Step 1 of `PrioritizeSuite`.

next test case is chosen by selecting the test case with the highest contribution value. However, if all test cases' contribution values are zero, we reset all coverage flags for the program entities and select the next test case to add to $orderedTestSuite$ based on the highest entity coverage (lines 11-13). The test case chosen, $nextTest$, is removed from $allTests$ and placed into the next available spot in $orderedTestSuite$, and all entities that are covered by $nextTest$ are marked `covered` (lines 17-19). This process iterates until all test cases in $allTests$ are consumed and, thus, $orderedTestSuite$ contains all of the test cases in the original test suite. Finally, $orderedTestSuite$ is returned.

For our example, Table 4 shows the additional coverage of the test cases in $allTests$. Test case T5 has the highest score and is thus chosen as the next test case in $orderedTestSuite$. Fig. 12 shows the state of the example after T5 is placed in $orderedTestSuite$ and removed from $allTests$. The slashed conditions represent conditions that are covered by test cases in $orderedTestSuite$. After two more iterations of the statements in Step 2, all conditions are covered as shown in Fig. 13. Table 5 shows that all scores for the additional entities covered are zero. Thus, all entities are marked uncovered, as shown in Fig. 14, and the next test case is chosen based on the number of entities covered. Finally, after five iterations of Step 2, $allTests$ is empty, and the $orderedTestSuite$ is ordered: T1, T5, T3, T4, T6, T2.

4.4 New Test-Suite Reduction Algorithm (Build-Up)

We produced a second test-suite reduction algorithm by extending the prioritization algorithm presented in Section 4.3 to detect when $orderedTestSuite$ provides

TABLE 3
Test Cases and Associated Contribution Weights Assigned during Step 1 of `PrioritizeSuite`

Test case	Entities Covered
T1	4
T2	2
T3	4
T4	3
T5	3
T6	4

TABLE 4
Test Cases and Associated Contribution Weights Assigned During the First Iteration of Step 2 of `PrioritizeSuite`

Test case	Additional Entities Covered
T1	-
T2	0
T3	1
T4	1
T5	2
T6	1

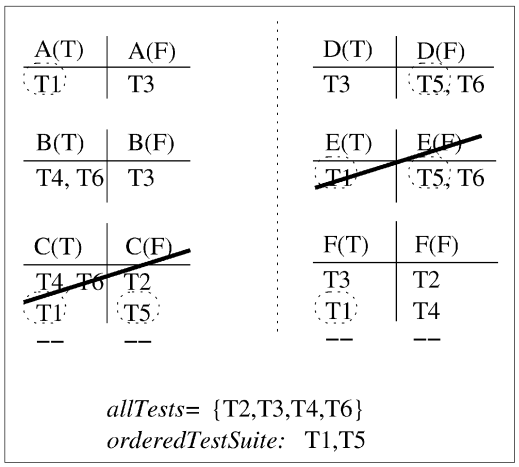

Fig. 12. Example after the first iteration of Step 2 of `PrioritizeSuite`.

coverage of the original test suite. This reduction algorithm is a build-up, additional technique that computes contributions in the same manner as our prioritization algorithm. The comment added after line 11 in Fig. 9 represents the extension to the prioritization algorithm to achieve a build-up reduction algorithm. At this point in the algorithm, if the greatest contribution value for all of the test cases in the original test suite is 0 and if all conditions, entries, exits, and cases are marked covered, the current set of test cases in *orderedTestSuite* at that point is a reduced test suite. If reduction were the goal instead of prioritization, execution of the algorithm could halt there. For example, Fig. 13 shows the state of the example at the point where *orderedTestSuite* can be used as a reduced suite.

5 Empirical Studies

This section describes a set of studies to evaluate the effectiveness of our test-suite reduction and test-suite prioritization algorithms.

To investigate the effectiveness of our algorithms presented in Section 4, we implemented prototypes of the reduction and prioritization algorithms. All three prototypes are written in C++. Much of the code is used in all prototypes because they utilize the same data structures. The combined code for the prototypes consists of 4,780 lines of code. We performed the studies on a Pentium III, 733 MHz computer.

TABLE 5
Test Cases and Initial Associated Contribution Weights Assigned during the Fourth Iteration of Step 2 of `PrioritizeSuite`

Test case	Additional Entities Covered
T1	-
T2	0
T3	-
T4	-
T5	-
T6	0

We used two subjects for our study: `TCAS` and `Space`. `TCAS` is a program that is used by airplanes to detect whether neighboring airplanes pose a danger of being too close. The component of the program that we are using suggests a solution to dangerous situations. This program, versions, and test cases were assembled by researchers at Siemens Corporate Research for a study of the fault-detection capabilities of control-flow and data-flow coverage criteria [8]. `TCAS` consists of 138 executable lines of C code. We used 41 faulty versions of the program, each containing one fault, and the *base* version, which is assumed to have no faults. The researchers at Siemens created the faulty versions by manually seeding `TCAS` with faults, usually by modifying a single line of code. Their goal was to introduce faults that were as realistic as possible based on their experience with real programs.

`Space` functions as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements and checks the contents of the file for adherence to the ADL grammar and to specify consistency rules. If the ADL file is correct, the `Space` program outputs an array data file containing a list of array elements, positions, and excitations; otherwise, the program outputs error messages. `Space` consists of 6,218 executable lines of C code. We also used 35 faulty versions of the

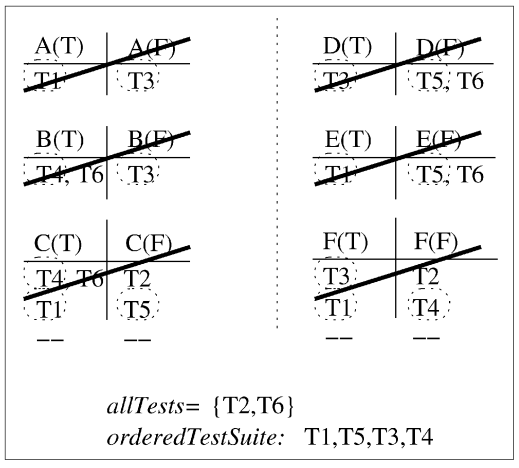

Fig. 13. Example after the third iteration of Step 2 of `Prioritize Suite`.

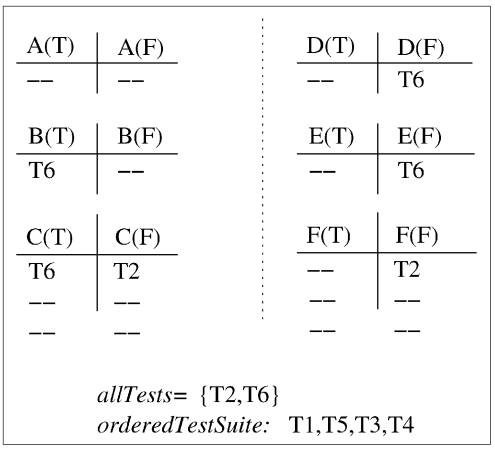

Fig. 14. Example during fourth iteration of Step 2 of `PrioritizeSuite` after all entities are again marked `uncovered`.

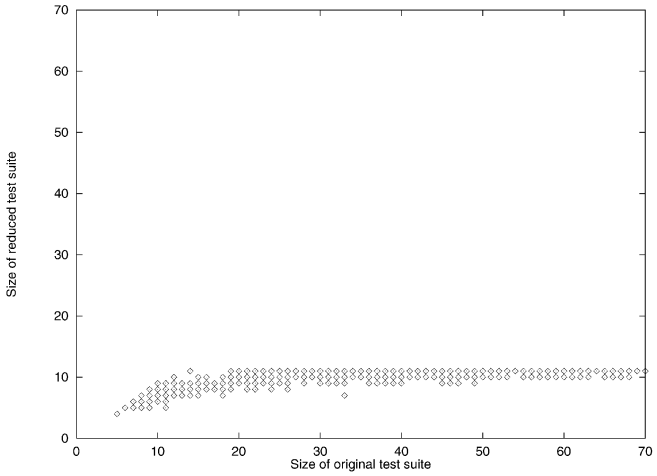

Fig. 15. Sizes of the reduced test suite versus the sizes of the original test suite for the break-down reduction method for TCAS.

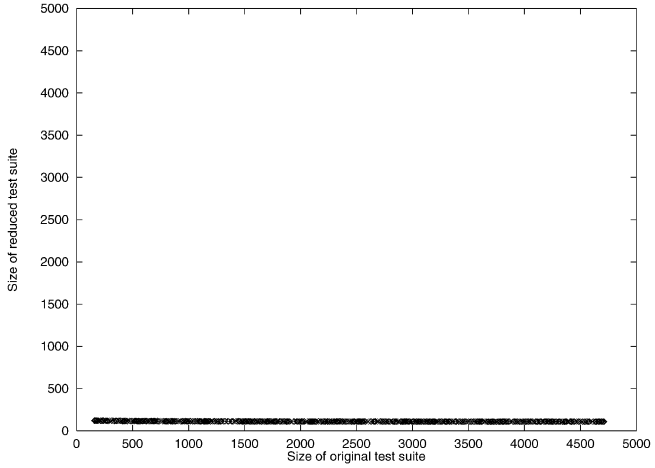

Fig. 16. Sizes of the reduced test suite versus the sizes of the original test suite for the break-down reduction method for Space.

program, each containing one fault and the *base* version, which is assumed for purposes of studies to contain no faults. Thirty of the faulty versions contain a single fault that had been discovered during the program's development. Through working with Space, Rothermel et al. [13] discovered an additional five faults and created versions with those faults.

We instrumented both TCAS and Space by hand for MC/DC by placing a probe at every condition, procedure entry, procedure exit, and case statement in a **switch**. For TCAS, we have a test pool of 1,608 test cases, and, for Space, we have a test pool of 13,585 test cases. From these test pools, 1,000 randomly sized, randomly generated (plus additional test cases to reach near-decision coverage) test suites, for each subject program, were extracted. These subjects and these test suites have been used in similar studies (e.g., [4], [11], [12]); these references provide additional details about the subjects. These test suites are near decision-coverage-adequate—only infeasible or extremely difficult to execute branches (such as those controlling an out-of-memory error condition) were not executed. For example, for Space, these test suites covered 80.7 to 81.6 percent of the 539 conditions in the 489 decisions. The test suites for TCAS ranged in size from 5 to 70 test cases. The test suites for Space ranged in size from 159 to 4,712 test cases.

We present three studies. The first evaluates the effectiveness and performance of our break-down test-suite reduction technique. The second evaluates the effectiveness and performance of our build-up test-suite reduction technique. The third evaluates the performance of our prioritization technique. Note that the order of the studies of the techniques differs from the order in which they were presented in Section 4—this lets us more conveniently compare the two reduction algorithms.

5.1 Break-Down Test-Suite Reduction Study

To evaluate our break-down test-suite reduction algorithm presented in Section 4.2, we applied our break-down test-suite reduction prototype for MC/DC to each of the 1,000 test suites for each subject.

For each test case in the union of all test suites (1,608 test cases for TCAS and 13,585 test cases for Space), we executed the subject program to acquire a test-case requirement coverage report. We then executed our algorithm with the coverage information for all test cases in each of the test suites. We evaluated three characteristics of the break-down test-suite reduction algorithm: size reduction of test suites, time to perform the reduction, and fault-detection loss of the reduced test suite.

To evaluate the break-down reduction technique's ability to reduce test suites, we plotted the size of the reduced test suites as a function of the original test suites. Figs. 15 and 16 are scatter plots for TCAS and Space, respectively. Each point in these scatter plots represents a test suite that has been reduced. Each point is positioned along the horizontal axis based on its original test-suite size and along the vertical axis based on its reduced test-suite size. Fig. 15 shows that the sizes of the reduced test suites for TCAS are fairly constant, although there is a slight dip for the smaller-sized original test suites. This dip is created because the smaller original test suites provided less MC/DC coverage than the larger ones; thus, the resulting reduced test suites were smaller. For TCAS, the average size of the reduced test suites is 9.9, the standard deviation is 1.5, the smallest reduced test suite contains four test cases, and the largest reduced test suite contains 11 test cases. Fig. 16 shows that the sizes of the reduced test suites for Space are nearly constant, regardless of the size of the original test suite. For Space, the average size of the reduced test suites is 111.8, the standard deviation is 3.4, the smallest reduced test suite contains 106 test cases, and the largest reduced test suite contains 125 test cases.

To evaluate the break-down reduction technique's performance in reducing test suites, we plotted the time to reduce each test suite as a function of the original test-suite size. Figs. 17 and 18 are scatter plots that show the time required to reduce the test suites for TCAS and Space, respectively, based on our break-down test-suite reduction algorithm. Each point in these scatter plots is positioned along the horizontal axis based on its original test-suite size and along the vertical axis based on the time needed by the

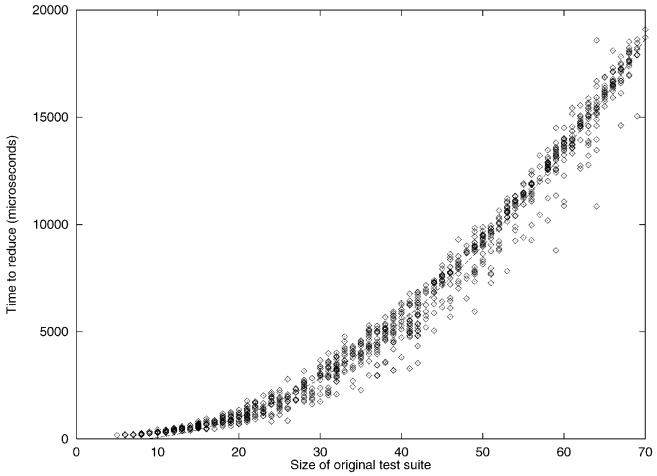

Fig. 17. Time to run the reduction algorithm versus the sizes of the original test suite for the break-down reduction method for TCAS.

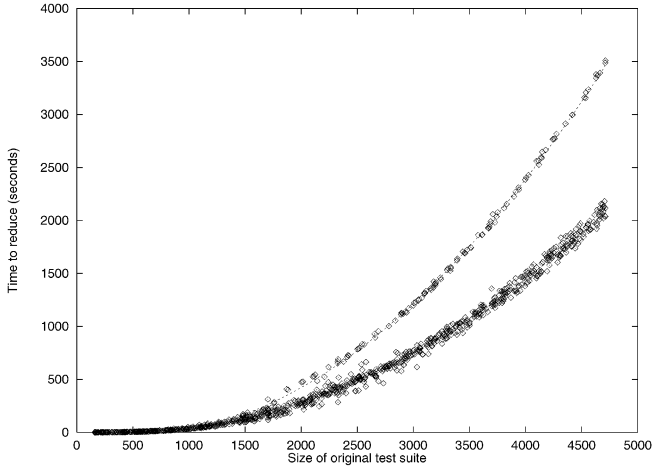

Fig. 18. Time to run the reduction algorithm versus the sizes of the original test suite for the break-down reduction method for Space.

prototype to produce the reduced test suite. Both figures show a quadratic curve. Note that the vertical axis for Fig. 17 represents time in microseconds, whereas the vertical axis for Fig. 18 represents the time in seconds. Fig. 17 shows a more diffuse quadratic curve for TCAS. We suspect, however, that this is due to the sensitivity of such small measurements—small variances such as the operating system's scheduling of processes can cause such results. Fig. 18 shows two distinct, more localized, quadratic curves for Space. To verify this quadratic trend, we computed the best-fit trendlines for the curves. Let x represent the size of the original test suite and y represent the time to reduce (in the units used for each figure). The best-fit trendline for Fig. 17 is $y = 4.32x^2 - 36.03x$ with an R^2 value of 0.9865. The best-fit trendline for the lower curve in Fig. 18 is $y = 0.00011x^2 - 0.082x$ with an R^2 value of 0.9975, and the best-fit trendline for the higher curve is $y = 0.00019x^2 - 0.1747x$ with an R^2 value of 0.9983.

An interesting phenomenon about Fig. 18 is the two modes of the curve. We further investigated this data to understand why this occurred. Upon analysis of the data, we discovered the cause of the diverging curves is related to the presence or absence of essential test cases in the original, unreduced test suite. Each unreduced test suite represented in the bottom curve had at least one essential test case. Each unreduced test suite represented in the top curve had no essential test cases. Because the essential test cases are removed from the problem space ($allTests$ in the algorithm), as well as every coverage entity ($allConditions$ in the algorithm) that those test cases covered, the problem space is immediately much smaller and, thus, the algorithm runs faster. In fact, about 30 percent of the coverage entities were covered by the essential test case or cases in each test suite in the lower curve. We identified truth values of three conditions in Space that contributed to this phenomenon. These three truth values are covered by only a few test cases in our test pool. Those test suites that contained exactly one of the test cases that covered at least one of those sparsely covered truth values reduced faster than those that had zero, two, or more of such test cases.

Because studies have shown that there is fault-detection loss in a reduced test suite [11], we also considered the fault-detection capability lost by each test suite when that test suite was reduced using our algorithm. To do this, we compared the ability of the reduced test suite to detect faults in 41 faulty versions of TCAS and in 35 faulty versions of Space. F denotes the number of distinct faults revealed by original test suite T over the faulty versions of program P, and F_{red} denotes the number of distinct faults revealed by reduced test suite T_{red} over those versions. The *number of faults lost* is given by $(F - F_{red})$, and the *percentage reduction in fault-detection effectiveness* of test-suite reduction is given by $(\frac{F-F_{red}}{F} * 100)$.

To evaluate the fault-detection loss of the reduced test suite compared to the original test suite, we plotted the percentage of fault-detection loss as a function of the original test-suite size. Figs. 19 and 20 are scatter plots that show the percentage reduction in fault detection of the reduced test suite compared to the original test suite for TCAS and Space, respectively. Each point in these scatter plots is positioned along the horizontal axis based on its original test-suite size and along the vertical axis based on its percentage of fault detection that was lost by the reduced test suite. Fig. 19 shows that the fault-detection loss varies greatly for TCAS, although there is a weak trend that more fault-detection loss occurs for larger test suites. This trend is caused by the fact that the largest test suites have the most reduction. For TCAS using the break-down test-suite reduction technique, the average fault-detection loss is 44.4 percent, the standard deviation is 21.0, the maximum fault-detection loss is 92.9 percent, and the minimum is 0 percent. These results are consistent with the results reported by Rothermel et al. [11]. Fig. 20 also shows varying fault detection loss and a weak trend of less fault-detection loss for small original test suites for Space. However, the overall fault-detection loss is much less than that of Fig. 19. The average fault-detection loss is 10.2 percent, the standard deviation is 5.1, the maximum fault-detection loss is 22.9 percent, and the minimum fault-detection loss is 0 percent. An interesting characteristic of Fig. 20 is the appearance of the stratification of the data. Upon investigation of the data, we have found that all 1,000 original test

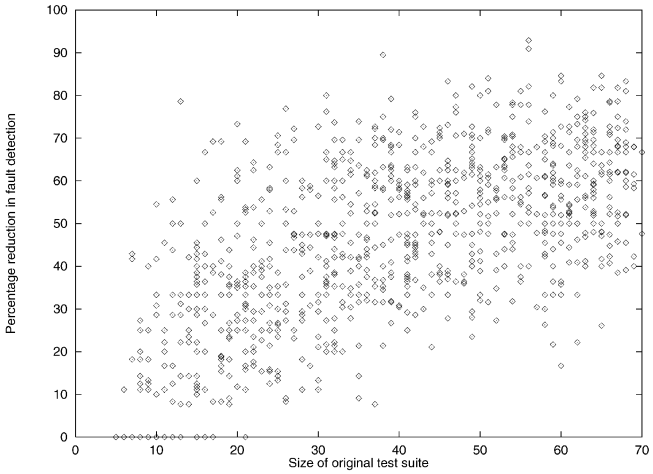

Fig. 19. Percentage reduction in fault detection as a result of the break-down reduction method for TCAS.

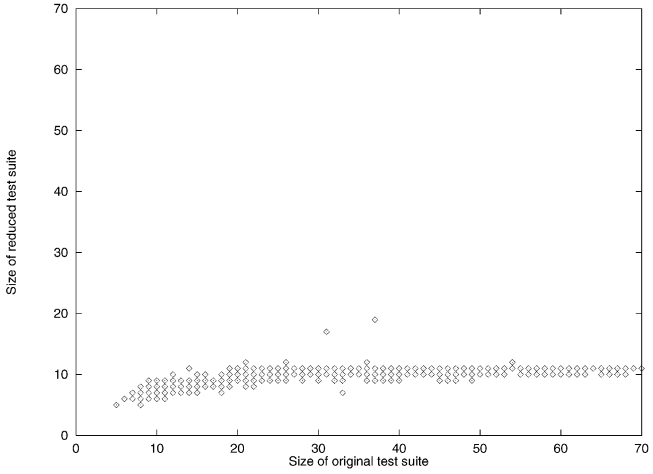

Fig. 21. Sizes of the reduced test suite versus the sizes of the original test suite for the build-up reduction method for TCAS.

suites detect 31 or more faults, with most detecting all 35 faults. The reduced test suites detected most of the faults detected by the original test suite with a difference of only a few missed faults. The stratification appears in the scatter plot because of the limited number of distinct values for the number of faults found in both the original and reduced test suites.

5.2 Build-Up Test-Suite Reduction Study

To evaluate our build-up test-suite reduction algorithm presented in Section 4.4 and compare its performance and effectiveness with our break-down reduction algorithm, we applied our build-up reduction prototype for MC/DC to each of the 1,000 test suites for each subject. Like the break-down test-suite reduction algorithm, we evaluated three characteristics of the build-up test-suite reduction algorithm: size reduction of test suites, time to perform the reduction, and fault detection loss of the reduced test suite.

To evaluate the build-up reduction technique's ability to reduce test suites, we plotted the size of the reduced test suites as a function of the original test suites. Figs. 21 and 22 are scatter plots, for TCAS and Space, respectively. Fig. 21 shows that the sizes of the reduced test suites are fairly constant for TCAS, with a slight dip for the smaller-sized original test suites. This result is consistent with Fig. 15 for break-down reduction. For TCAS, the average size of the reduced test suites is 10.1, the standard deviation is 1.4, the smallest reduced test suite contains five test cases, and the largest reduced test suite contains 19 test cases. These results are only slightly higher (1.1 percent on average) than those of the break-down study. Fig. 22 shows that the sizes of the reduced test suites are nearly constant for Space, again, like the results of the break-down study. For Space, the average size of the reduced test suite is 116.4, the standard deviation is 3.4, the smallest reduced test suite contains 108 test cases, and the largest reduced test suite contains 129 test cases. These results are 4.1 percent higher than those of the break-down study.

To evaluate the build-up reduction technique's performance in reducing test suites, we plotted the time to reduce each of the test suites as a function of the original test-suite size. Figs. 23 and 24 are scatter plots, for TCAS and Space, respectively, that show the time required to perform the build-up reduction technique using our prototype. Both figures exhibit a linear trend. Note that the vertical axis for Fig. 23 represents microseconds, whereas the vertical axis

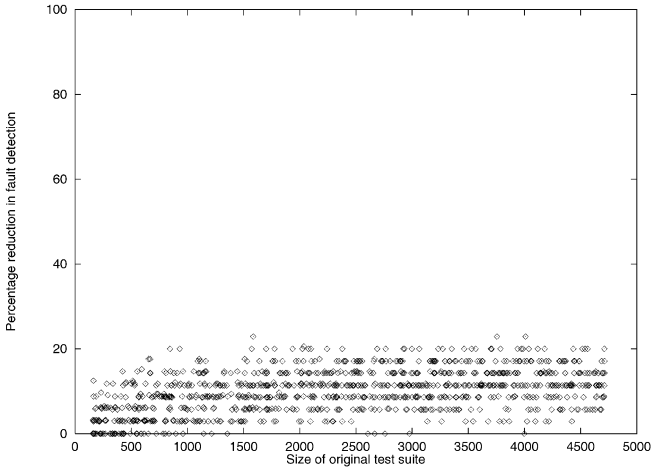

Fig. 20. Percentage reduction in fault detection as a result of the break-down reduction method for Space.

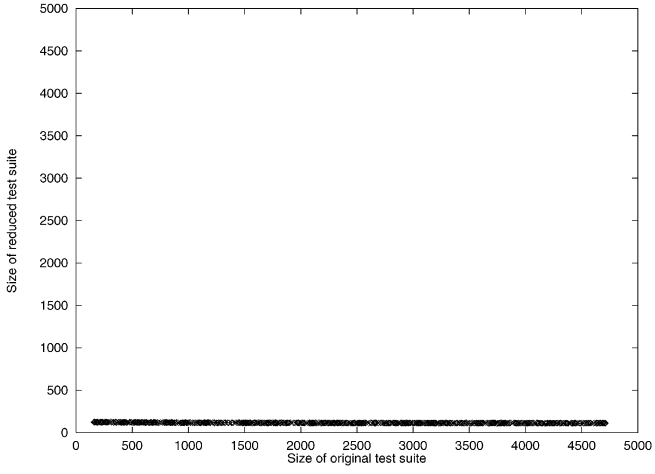

Fig. 22. Sizes of the reduced test suite versus the sizes of the original test suite for the build-up reduction method for Space.

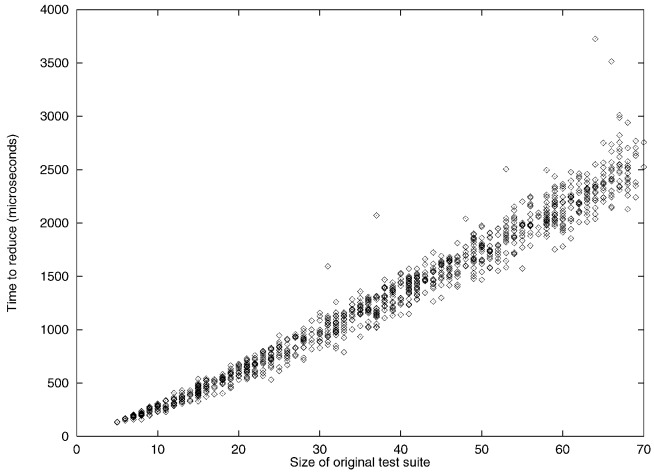

Fig. 23. Time to run the reduction algorithm versus the sizes of the original test suite for the build-up reduction method for TCAS.

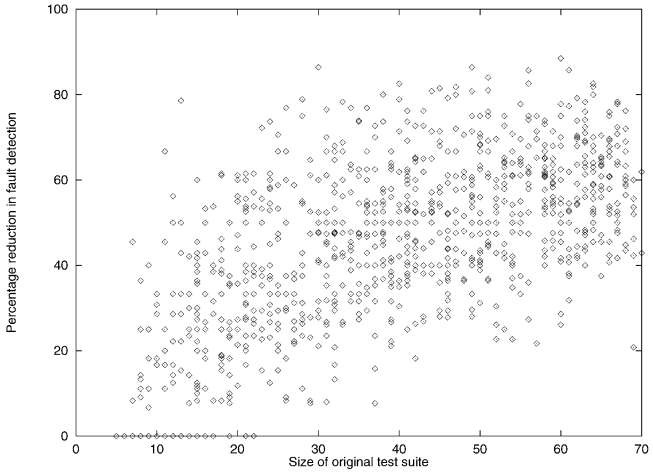

Fig. 25. Percentage reduction in fault detection as a result of the build-up reduction method for TCAS.

for Fig. 24 represents seconds. Fig. 23 shows a more diffuse line for TCAS than Fig. 24. We suspect that this diffuseness is due to the sensitivity of such small measurements. Fig. 24 shows a more localized line for Space. The times to reduce the test suites using the build-up reduction technique are much less than the times achieved in the break-down reduction study. The build-up reduction technique is much more efficient than the break-down reduction technique.

To evaluate the fault-detection loss of the reduced test suite compared to the original test suite, we plotted the percentage of fault-detection loss as a function of the original test-suite size. Figs. 25 and 26 are scatter plots that show the percentage reduction in fault detection of the reduced test suite compared to the original test suite for TCAS and Space, respectively. Fig. 25 shows varying fault-detection loss for TCAS. This result is consistent with both [11] and our break-down study. Using the build-up reduction technique, the average fault detection loss is 43.7 percent, the standard deviation is 21.1, the maximum fault detection loss is 88.5 percent, and the minimum fault-detection loss is 0 percent. Fig. 26 for Space also shows varying fault-detection loss, although its maximum is much less than that of TCAS. The average fault-detection loss is 9.9 percent, the standard deviation is 5.0, the maximum fault-detection loss is 22.9 percent, and the minimum fault-detection loss is 0 percent.

5.3 Prioritization Study

To evaluate the performance of our prioritization algorithm presented in Section 4.3, we applied our prioritization prototype for MC/DC to each of the 1,000 test suites for each subject. We evaluated the time to perform this prioritization for each of these test suites. Figs. 27 and 28 show the results of this study for TCAS and Space, respectively. Note that the vertical axis of Fig. 27 represents microseconds, whereas the vertical axis of Fig. 28 represents seconds. Both figures show a quadratic curve. The curve for the timings for TCAS is again more diffuse than for that of Space because of the sensitivity of the small timings for TCAS. To verify this quadratic trend, we computed the best-fit trendlines for the curves. Let x represent the size of the original test suite and y represent the time to prioritize (in the units used for each figure). The best-fit trendline for Fig. 27 is $y = 1.66x^2 + 2.45x$ with an R^2 value of 0.9789. The best-fit trendline for Fig. 28 is $y = 0.00010x^2 - 0.1747x$ with an R^2 value of 1.000 (to three decimal places).

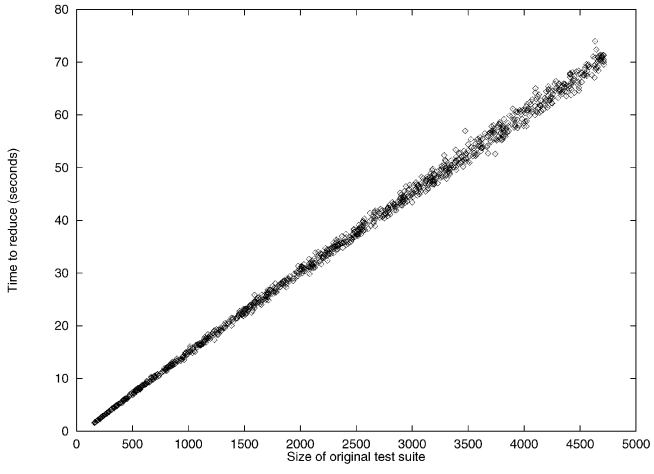

Fig. 24. Time to run the reduction algorithm versus the sizes of the original test suite for the build-up reduction method for Space.

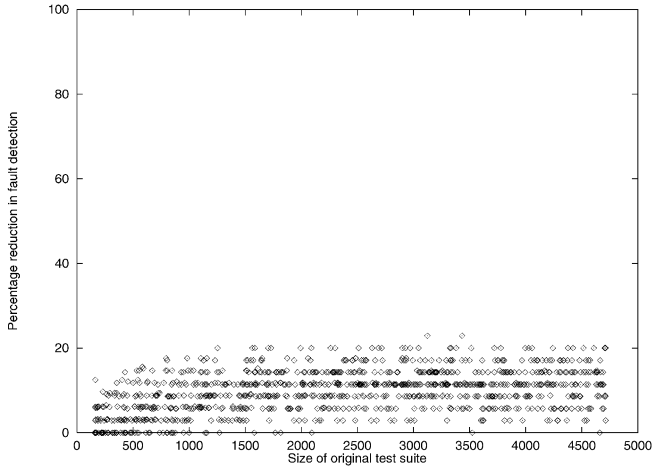

Fig. 26. Percentage reduction in fault detection as a result of the build-up reduction method for Space.

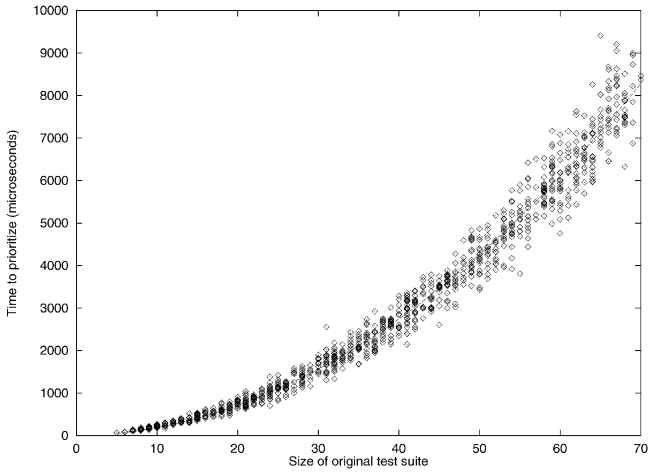

Fig. 27. Time to run the prioritization algorithm versus the sizes of the original test suite for `TCAS`.

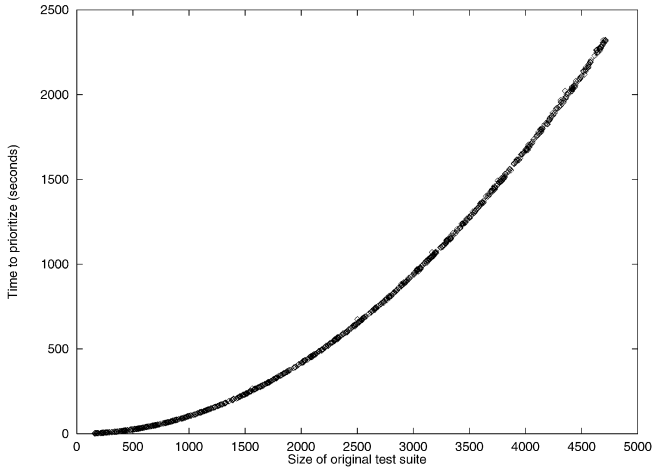

Fig. 28. Time to run the prioritization algorithm versus the sizes of the original test suite for `Space`.

6 Conclusions

This paper has presented two new algorithms for test-suite reduction and one new algorithm for test-suite prioritization that can account for MC/DC when reducing and prioritizing test suites. The paper also presents the results of empirical studies that evaluate these algorithms. The results achieved thus far are encouraging in that they show the potential for substantial test-suite size reduction with respect to MC/DC. Such techniques can significantly reduce the cost of regression testing for those users of this powerful testing criterion.

Our studies have shown that the size reduction of test suites can be substantial for both the break-down and build-up test-suite reduction techniques presented in Sections 4.2 and 4.4, respectively. For our subjects, the break-down test-suite reduction technique requires quadratic time to execute, whereas the build-up reduction technique requires linear time. The fault-detection loss was consistent between the two techniques. The size of the reduced test suite for the break-down technique was consistently lower than the size of the reduced test suite for the build-up technique.

The break-down technique offers two advantages over the build-up one: effectiveness of reduction and flexibility. First, the break-down technique produces reduced test suites that are consistently smaller than those produced by the build-up technique. Although the difference in reduced test-suite size is generally small, there are two situations where the extra time spent in the reduction of the test suite may be beneficial. The first situation is one where the test suite will be reduced once, and the reduced test suite will evermore be considered the full test suite. In this situation, the cost of the reduction can be amortized over the numerous executions of the test suite. The second situation exists when the cost of running each test case is high. One such domain is the testing of critical systems for commercial airborne systems, where test cases are performed on simulation equipment and require a large setup and are thus very expensive. Such domains may find that the additional expense of time spent in reducing the test suite is more than justified by the time saved during the execution of the test suite. Second, the break-down technique is more flexible because it lets a testing manager verify, and override, the omission of each test case as the technique executes. The ability to verify and override the omission of each test case may be important in settings where certain test cases are required for reasons such as functional-based testing or mandated test-case redundancy. The test-suite reduction can be stopped at any time, and the remaining test cases (that have not been discarded) can be taken as a reduced test suite that has the same coverage as the original test suite.

The build-up technique offers the advantage that it is much faster than the break-down technique. This technique would be useful when the test suite being reduced is very large, thus taking the break-down technique too long. This technique would also be appropriate when the test cases in the test suite are very inexpensive and, thus, the cost of the potentially larger reduced test suite is inconsequential.

We provided the studies that determine the fault-detection loss to demonstrate the effectiveness of test suites that are reduced using MC/DC as the testing criterion. For each of our reduction techniques, the fault-detection loss can vary greatly based on the program and test suites. The consistency of the fault-detection loss between our two techniques and those presented by Rothermel et al. [11] suggests that fault-detection loss is more of a reflection on the program under test (and its faults), the original test suite, and the coverage criterion used, than it is on the reduction technique. More specifically, a perfect reduction technique for a given coverage criterion would generate a minimal number of test cases that provide the same coverage as the original test suite, without regard to the loss in fault detection. Coverage criteria attempt to associate coverage of the program under test with the fault-detection ability of test suites that were developed to satisfy the coverage criteria. Any technique that may be developed for MC/DC should be expected to achieve similar results to ours if they approximate minimal coverage (which is the goal of reduction). It should be noted that the subject programs and test suites used were not MC/DC-adequate —the reduced test suites provided the same coverage based on MC/DC as the original ones, but the fault detection ability of reduced MC/DC-adequate test suites may perform more effectively in this regard.

The prioritization technique offers a way to order all test cases in a test suite. Prioritizing a test suite instead of reducing it is appropriate in two cases. First, if the tester

wishes to rerun all test cases in the test suite to avoid any fault-detection loss, the prioritization technique can give an ordered test suite in which the MC/DC coverage of the original suite is achieved quickly. Such an ordered test suite would potentially enable earlier discovery of failures. Second, if the tester has a limited, allotted time to test, then the ordered suite can be run until the allotted time expires. If the allotted time is less than the time required to run a reduced suite, running the ordered test suite until the allotted time expires should achieve more coverage than an unordered test suite. If the allotted time is greater than the time required to run a reduced suite, more test cases can be run to possibly achieve greater fault detection.

Although our initial studies are encouraging, much more experimentation must be conducted to verify the effectiveness of our techniques and MC/DC reduction and prioritization in general and in practice. Specifically, we are planning experiments that use our test-suite reduction and prioritization prototypes on commercial airborne software where MC/DC is required. These studies will let us further investigate the fault-detection capabilities of an MC/DC-adequate test suite on software for which MC/DC was designed. These studies will also let us evaluate our algorithms and help us provide guidelines for test-suite reduction and prioritization in practice.

ACKNOWLEDGMENTS

This work was supported in part by a grant from Boeing Aerospace Corporation to Georgia Tech, by US National Science Foundation awards CCR-9707792, CCR-9988294, CCR-0096321, and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the `Space` program and many of its test cases. Gregg Rothermel and the Galileo Research Group at Oregon State University helped in the preparation of the subject programs and test suites. The anonymous reviewers provided many helpful suggestions that greatly improved the empirical studies and the presentation of the results.

REFERENCES

[1] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135-141, Mar. 1996.

[2] Y. Chen, D. Rosenblum, and K. Vo, "TestTube: A System for Selective Regression Testing," *Proc. 16th Int'l Conf. Software Eng.*, pp. 211-222, May 1994.

[3] J.J. Chilenski and S.P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Eng. J.*, vol. 9, no. 5, pp. 193-200, 1994.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 102-112, Aug. 2000.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," *Proc. Int'l Conf. Software Eng.*, pp. 329-338, May 2001.

[6] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman, 1979.

[7] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methods*, vol. 2, no. 3, pp. 270-285, July 1993.

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l. Conf. Software Eng.*, pp. 191-200, May 1994.

[9] J. Offutt, J. Pan, and J.M. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets," *Proc. 12th Int'l Conf. Testing Computer Software*, pp. 111-123, June 1995.

[10] G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Software Eng. and Methods*, vol. 6, no. 2, pp. 173-210, Apr. 1997.

[11] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," *Proc. Int'l Conf. Software Maintenence*, pp. 34-43, Nov. 1998.

[12] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.*, vol. 27, no. 10, pp. 929-948, Oct. 2001.

[13] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization: An Empirical Study," *Proc. Int'l Conf. Software Maintenence*, pp. 179-188, Sept. 1999.

[14] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software—Practice and Experience*, vol. 28, no. 4, pp. 347-369, Apr. 1998.

[15] W.E. Wong, J.R. Horgan, A.P. Mathur, and A. Pasquini, "Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application," *Proc. 21st Ann. Int'l Comp. Software and Application Conf.*, pp. 522-528, Aug. 1997.

James A. Jones received the BS degree in computer science from the College of Engineering at Ohio State University. He is currently a PhD student in the College of Computing at the Georgia Institute of Technology, Atlanta. From 1997 to 2000, he was a research scientist at Ohio State University and then at the Georgia Institute of Technology. His research interests include program analysis, visualization, and testing. To date, his research has investigated the maintenance of test suites, as well as the use of software visualization to aid in the task of fault localization. He is a member of the ACM and ACM SIGSOFT.

Mary Jean Harrold received the BS and MA degrees in mathematics from Marshall University and the MS and PhD degrees in computer science from the University of Pittsburgh, Pennsylvania. She is the US National Science Foundation ADVANCE Professor of Computing and associate professor in the College of Computing at the Georgia Institute of Technology, where she is a member of the Center for Experimental Research in Computer Systems (CERCS) and the Graphics, Visualization, and Usability Center (GVU-Center). Her research interests include the development of efficient techniques and tools that will automate, or partially automate, development, testing, and maintenance tasks. Dr. Harrold is a recipient of the US National Science Foundation's National Young Investigator Award for her work in regression testing and object-oriented analysis and testing. She serves on the editorial boards of the *IEEE Transactions on Software Engineering*, *ACM Transactions on Programming Languages and Systems*, and *Empirical Software Engineering Journal*. She served as program cochair for the 23rd International Conference on Software Engineering 2001 and as program chair for the ACM SIGSOFT International Symposium on Software Testing and Analysis 2000 and for the IEEE International Conference on Software Maintenance 1997. Dr. Harrold is a member of the Computing Research Association's Committee on the Status of Women in Computing (CRA-W), where she directed the committee's Distributed Mentor Project. She is cochair of CRA-W. She is a member of the ACM, IEEE Computer Society, and Sigma Xi.