Automatically Generating Test Data from a Boolean Specification

Elaine Weyuker, Tarak Goradia, and Ashutosh Singh

Abstract—**This paper presents a family of strategies for automatically generating test data for any implementation intended to satisfy a given specification that is a Boolean formula. The fault detection effectiveness of these strategies is investigated both analytically and empirically, and the costs, assessed in terms of test set size, are compared.**

Index Terms—**Automatic test case generation, black-box testing, software testing**

I. Introduction

LEVESON *et al.* [8] describe a formal specification technique that uses a modified Statechart notation to specify TCAS II, an aircraft collision avoidance system. This specification represents an example of a formally specified real-world process-control system. Motivated by this work, we used the AND–OR table representation of the conditions for state transitions in this specification as the basis for designing algorithms to automatically generate test cases. Because these tables are simply one way of representing a Boolean formula, we investigated the advantages of having such a regimented form of a specification from the perspective of automatic generation of test data.

Even for a formal specification of such rigid and simple format, exhaustive testing is typically prohibitively expensive, because a formula of n variables would require 2^n distinct test cases—too many for any formula of even modest complexity, especially when one considers the number of formulas that comprise the specification of an entire system. Therefore, our goal was to define algorithms to automatically generate test sets that would be substantially smaller than exhaustive test sets, but would nonetheless be highly effective at detecting faults. Once that was accomplished, we developed a tool to implement these algorithms and devised an empirical study to determine how well the algorithms worked using selected parts of the TCAS II specification.

An important question we addressed is, How should we evaluate effectiveness? This is a difficult issue that has been addressed in a number of recent papers [4], [6], [10], [11], [12]. Although the importance of this question has begun to be acknowledged, there is, at this time, far from universal agreement about the answer. In this paper, we use mutation analysis [2] as the basis for this assessment of effectiveness.

Mutation analysis is a fault-based testing strategy that starts with a program to be tested and makes numerous small syntactic changes to it, creating a set of *mutant programs*. Each mutant is then run on a test set that is being evaluated to see whether the test data are comprehensive enough to distinguish the original program from each inequivalent mutant. The intuition is that each such mutant represents a "buggy" version of the program, and, if the test set cannot distinguish the two versions (i.e. if both versions produce the same output for every input), then the inserted bug (mutation) would go undetected by the test set. If many mutants are not distinguished by the test set, then many bugs would go undetected. If almost all mutants are distinguished, then almost all *of the bugs represented by the mutations* would be detected. If the mutations really represent "typical" faults, then a high ratio of distinguished mutants to total inequivalent mutants would indicate an effective testing strategy. This ratio $\times$ 100 is known as the *mutation score*.

In Section III, we present a basic testing technique for generating test data for programs that implement specifications represented by Boolean formulas. This basic strategy is nondeterministic. Section IV investigates the fault detection ability of the basic strategy analytically. In Section V, we present a family of testing strategies that represent enhancements of the basic strategy. In Section VI, we describe our experience in using these strategies to test parts of the TCAS II specification. We have selected 20 of the larger Boolean formulas from the specification, and have used the tool we built to automatically generate test sets using each of the new strategies. Our empirical results are summarized in that section. Section VII presents our conclusions.

II. Definitions

The following notation is used throughout this paper. $+$, $\cdot$, and $-$ represent the "or," "and," and "not," Boolean operations, respectively. Usually, the "$\cdot$" is omitted. We use 1 and 0 to denote "true" and "false," respectively. ϕ is used to denote a "don't care" value (i.e., a value that can be either 0 or 1).

A given Boolean formula F can be represented in various standard formats, including *sum-of-products* form or *product-of-sums* form. These are also known, respectively, as *disjunctive normal form* and *conjunctive normal form*. For example,

Manuscript received August 30, 1993; revised February 1994. This work was supported in part by the National Aeronautics and Space Administration (NASA) under Grant NAG–1–1238, and in part by the National Science Foundation (NSF) under Grant CCR–8920701. Recommended by J. D. Gannon.

E. J. Weyuker is with AT&T Bell Laboratories, Murray Hill, NJ 07974, USA, and the Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA. E-mail: weyuker@research.att.com.

T. Goradia is with Siemens Corporate Research, Princeton, NJ 08540, USA. e-mail: tarak@scr.siemens.com.

A. Singh is with the Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA.

IEEE Log Number 9400540.

consider the formula $a(b\bar{c}+\bar{d})$. In disjunctive normal form, this can be represented as $ab\bar{c}+a\bar{d}$. In conjunctive normal form, it can be represented as $(a)(b+\bar{d})(\bar{c}+\bar{d})$. These representations are not unique, but there exist canonical representations that are unique for a given Boolean formula up to commutativity. One such representation is known as *canonical disjunctive normal form*. If a product-term contains every variable of F, either in the complemented or uncomplemented form, it is called a *minterm*. The canonical disjunctive normal form of formula F is the unique sum-of-products representation of F in which each product-term is a minterm. The canonical disjunctive normal form representation for the above formula is as follows:

$$ab\bar{c}d + ab\bar{c}\bar{d} + abcd + a\bar{b}cd + a\bar{b}\bar{c}\bar{d}.$$

Each occurrence of a variable or its negation in a Boolean formula is called a *literal*. A *test case* is an assignment of values to variables of a formula. A formula in disjunctive normal form is said to be *irreducible*, provided that none of its literals or terms can be deleted without altering the value of the formula for some test case. Let F be a Boolean formula in irreducible disjunctive normal form, containing n variables and m product-terms: $p_1+p_2+\cdots+p_m$. The ith product-term is denoted by $p_i = (l_{i,1}l_{i,2}\cdots l_{i,k_i})$, where $l_{i,j}$ denotes the jth literal in the ith term.

A literal occurrence in a Boolean formula is said to have *meaningful impact* on the value of the formula for a given test case if, everything else being the same,[1] a different truth value assignment to that literal would have resulted in the formula evaluating to a different value. We will sometimes speak of a test case *demonstrating* that the literal occurrence has meaningful impact on the specified output of a Boolean formula, meaning that the literal occurrence has meaningful impact on that output for that test case. For example, consider the formula $(ab + ac)$. To facilitate the discussion, we refer to the first occurrence of the variable a as a_1, and the second occurrence as a_2. Consider the test case $a = 0, b = 1, c = 0$. This causes the formula to evaluate to 0. If a_1 had the value 1, however, the formula would have evaluated to 1. Hence, a_1 has meaningful impact on the 0 value for this test case. However, this test case does not demonstrate that b, a_2 or c have meaningful impact on the 0 value, because a change in the value of any one of them alone would not change the value of the expression. Consider another test case for this formula,$a = 0, b = 1, c = 1$, which also causes it to evaluate to 0. Both a_1 and a_2 have meaningful impact on the 0 value for this test case, because a change in the value of either one would cause the value of the expression to change from 0 to 1. Our strategies involve the selection of test cases that demonstrate the meaningful impact of each literal occurrence on each possible value of the formula.

To test an implementation of a given formula, test cases have to be picked from n-dimensional Boolean space, where n is the number of distinct variables in F. Test cases generated by our strategies fall into two general categories: true points and false points. *True points* are those that cause the formula to evaluate to 1. We denote the set of all true points by R. *False points* are those that cause the formula to evaluate to 0. We denote the set of all false points by R'.

We first consider true points. Letting p_i denote the ith product-term in the disjunctive normal form representation of the formula F, the points of the input space that cause p_i to evaluate to 1 constitute the *true points* associated with p_i. We denote these points by R_i. Since the sum-of-products form is assumed to be irreducible, the true points associated with a given term are not a subset of the true points associated with any other term.

In the remainder of this section, we consider the Boolean formula $(a(\bar{b} + \bar{c})d + e)$ to illustrate our terminology. This formula can be represented in irreducible disjunctive normal form as $a\bar{b}d + a\bar{c}d + e$. Note that $a\bar{b}d$ represents four minterms: $a\bar{b}cde$, $a\bar{b}cd\bar{e}$, $ab\bar{c}de$, and $ab\bar{c}d\bar{e}$. Each of these minterms can be represented by a five-digit binary number, with a 0 representing a negated variable and a 1 representing a non-negated variable. Then these four minterms can be represented as $10111, 10110, 10011, 10010$, respectively, or, using decimal notation, 23, 22, 19, and 18. These four points are the true points for $a\bar{b}d$. Similarly, the true points for $a\bar{c}d$ can be represented by 18, 19, 26, and 27, and the true points for e can be represented by $1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29$, and 31.

Note that the union of the sets of all the true points of the product-terms of F is simply the set of true points of F. The true points for our example formula are therefore represented by the set $\{1, 3, 5, 7, 9, 11, 13, 15, 17, 18, 19, 21, 22, 23, 25, 26, 27, 29,$ and $31\}$. This is just the minterm representation of the original formula $F = (a(\bar{b} + \bar{c})d + e)$.

We distinguish between two types of true points for a given term p_i of F. The *unique true points* for the term p_i are those points that are in R_i, but do not belong to any other R_j. We denote these points by U_i. These points are of interest because they demonstrate the meaningful impact of each literal of a term on the evaluation of the formula to true. Each of the strategies that we introduce in Section V, therefore, require the selection of at least one point from the set of unique true points associated with each term.

There is one unique true point for $a\bar{b}d$, namely, 22, because 18 and 19 are also true points for the term $a\bar{c}d$, and 23 is a true point for e. There is also just one unique true point for $a\bar{c}d$: 26. For e, the unique true points are $1, 3, 5, 7, 9, 11, 13, 15, 17, 21, 25, 29$, and 31.

The other type of true point is known as an *overlapping true point*. We denote this set of points by O. O consists of those points that are true points for at least two terms. Thus, the set of true points for a formula can be viewed as consisting of disjoint subsets $U_1, \cdots, U_m, O$. For our example, O consists of the points 18, 19, 23, and 27.

We next define different types of false points. The false points of F can be represented by the set $\{0, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 30\}$ for our example.

Let $p_{i,j}$ denote the product-term obtained by complementing the jth literal of the product-term p_i. For example, if p_i is $a\bar{b}d$,

[1] All other literal occurrences except the one under examination have the same values as during the original evaluation.

then $p_{i,3}$ is $a\bar{b}\bar{d}$. Let k_i be the number of literals in term p_i, and let n be the number of distinct variables in the formula F. We denote the set of true points for $p_{i,j}$ by $D_{i,j}$. The number of points in such a set is 2^{n-k_i}, corresponding to all combinations of the values of the $(n-k_i)$ variables missing in the product-term p_i, denoted $p_{i,1}, \cdots, p_{i,k_i}$. Since the product-term p_i has k_i literals, there are k_i such sets associated with p_i.

For example, consider again the formula $a\bar{b}d+a\bar{c}d+e$, which contains five variables: a, b, c, d, and e. The first product-term $p_1 = a\bar{b}d$ contains three literals $a, \bar{b}$, and d, with $p_{1,1} = \bar{a}\bar{b}d, p_{1,2} = abd$, and $p_{1,3} = a\bar{b}\bar{d}$. Each of the associated sets contains $4 = 2^{5-3}$ points, corresponding to the four possible combinations of truth value assignments to c and e. $D_{1,1}$ contains the points $2, 3, 6$, and 7. $D_{1,2}$ contains the points $26, 27, 30$, and 31, and $D_{1,3}$ contains the points $16, 17, 20$, and 21. Similarly, $D_{2,1}$ contains the points $2, 3, 10$, and 11, $D_{2,2}$ contains $22, 23, 30$, and 31, and $D_{2,3}$ contains $16, 17, 24$, and 25. $D_{3,1}$ consists of the points $0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28$, and 30.

Notice that the points in $D_{i,j}$ are true points for $p_{i,j}$, but may be either true points or false points for the formula F. In our example, $D_{1,1}$ consists of two true points for F (3 and 7) and two false points for F (2 and 6). We now want to select those points in $D_{i,j}$ that are false points for F. We denote that set by $N_{i,j}$. For our example, $N_{1,1}$ contains 2 and 6, $N_{1,2}$ contains 30, $N_{1,3}$ contains 16 and 20, $N_{2,1}$ contains 2 and 10, $N_{2,2}$ contains $30, N_{2,3}$ contains 16 and 24, and $N_{3,1} = R' = \{0, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 30\}$.

For a given product-term, p_i, the union of all the $N_{i,j}$'s is denoted by N_i, and will be called the *near false points for* p_i. For our example, N_1 contains the points $2, 6, 16, 20$, and 30.

Finally,

$$N = \cup_{i=1}^{m} N_i = R' \bigcap \left(\cup_{i=1}^{m} \cup_{j=1}^{k_i} D_{i,j} \right).$$

We will call N the set of *near false points for* F. For our example formula, $N = \{0, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 30\}$.

The *remaining false points for* F are those false points that are not in N. We denote this set by M. Informally, there are false points that are close to true points (N), and those that are not (M). For our example, M is empty.

III. The Basic Strategy

In this section, we present the intuition underlying the basic meaningful impact strategy, and discuss several related proposed strategies. We present examples to illustrate the operation of the strategy, and finally a formal description of the strategy for Boolean specifications in irreducible disjunctive normal form.

The intuition underlying our approach is that given a representation of a Boolean formula, a test set should be chosen such that each literal occurrence in the representation demonstrates its meaningful impact on the outcome, if that is possible. Viewed another way, we argue that whenever a test set exists that does not contain any test cases that demonstrate the meaningful impact on the value of the formula for a given literal occurrence, an unnegated literal can be changed to a negated one, or a negated literal can be changed to an unnegated one, without noticeable effect. Therefore, if the implementation is wrong in this way, the fault would go undetected. In a sense, this strategy is testing directly for one particular type of fault that we call a *Variable Negation Fault.*

There have been several other testing approaches, used in different settings, that use the same underlying approach. The best known of these are the algorithms traditionally used to detect so-called stuck-at-0 and stuck-at-1 faults in combinational circuits. For that purpose, it is argued that it is reasonable to restrict attention to this class of faults because "most circuit failures fall into this class, and many other failures exhibit symptomatically identical effects" [7, p. 219].

Unfortunately, we do not have this type of data for software. One of the goals of our work, therefore, was to determine empirically whether test sets that are designed explicitly to guarantee the absence of variable negation faults will also detect, with high probability, other classes of faults. We examine that issue analytically in Section IV and empirically in Section VI.

In the software testing literature, Foster [3] proposed a similar algorithm for testing logical expressions in decision statements and assignment statements in implementations. His work was directly motivated by the earlier hardware approach. He argued that exhaustive testing is generally impractical and practically unnecessary, provided that test cases are selected that require "each variable value to individually affect the result" [3, p. 120] and determine that the operator is correct. His algorithm is completely deterministic, and represents one possible way of resolving the nondeterminism in our basic meaningful impact strategy. In particular, he suggested the following rule for avoiding nondeterminism: Although generating a test case to demonstrate the meaningful impact of literal ℓ, assign 1 to all free variables in "and/nor" relationship with ℓ and assign 0 to all free variables in "or/nand" relationship with ℓ.

In [9], Tai proposed a related strategy for testing conditional statements, provided that they contained only *singular Boolean expressions*. These are expressions that contain at most one occurrence of any variable. Very positive empirical results are reported in that paper, but only for Boolean expressions constrained in this way. In addition, the empirical evaluation considered only expressions containing three or four variables and did the assessment in terms of *any* other singular Boolean expression containing the same set of variables. Although the mutation scores reported in [9] are impressive, they are tempered by the severe limitation to singular expressions, and by the fact that arbitrary faults were included, with no evaluation of whether they represent faults that might plausibly be made.

We found that when we assessed our test generation strategies by using arbitrary faults rather than simple faults (represented by mutation operators), we always had a 100% fault detection rate. That is, when we randomly generated Boolean formulas and viewed them as faulty implementations of a given Boolean specification, all of our strategies detected all of the inequivalent formulas. This is not surprising, because it is as if programmers were told to implement a specification knowing only the name and number of variables. Obviously,

TABLE I
TEST CONDITIONS REQUIRED FOR THE FORMULA $ab(cd + e)$

Row#	Direct Effect		Test Condition				
	Of Literal	On Outcome	a	b	c	d	e
1	a	1	1	1	$cd+e=1$		
2	a	0	0	1	$cd+e\neq 1$		
3	b	1	1	1	$cd+e=1$		
4	b	0	1	0	$cd+e=1$		
5	c	1	1	1	1	1	0
6	c	0	1	1	0	1	0
7	d	1	1	1	1	1	0
8	d	0	1	1	1	0	0
9	e	1	1	1	$cd=0$		1
10	e	0	1	1	$cd=0$		0

the likelihood of implementing a correct program under such circumstances would be very low, and because the program would likely differ from the intended program in many places, even small test sets would be likely to detect the presence of some fault.

Our empirical results, reported in Section VI, are very encouraging and include Boolean formulas without the limitation to singular formulas. In addition, our empirical studies used substantially larger-size formulas than those considered by Tai. Also, our study is fault-based and considers the effectiveness of the strategies for detecting five particular classes of faults.

Others have recently reported using strategies that are related to those we describe here. For example, in [1], Chilenski and Miller discuss the modified condition and/or decision coverage criterion, which requires that "one must demonstrate that the outcome of a decision changes as a result of changing a single condition." This is used as an adequacy criterion to assess the comprehensiveness of a given test set. Gelperin [5] independently proposed the unique cause strategy, which he describes as being a "requirement-based strategy for choosing an operationally effective subset" to test complex decision logic.

In contrast to Foster's strategy, Tai's strategy, and the modified condition and/or decision criterion, our algorithms are explicitly designed to be used to automatically generate test sets to test *any* proposed implementation of a given Boolean *specification*, rather than to test a given individual *implementation*. Thus, our perspective is different than the above-cited work. In addition, we have defined six variants of the basic meaningful impact strategy, and have implemented a tool that automatically generates test sets for a given Boolean formula specification by using each of these algorithms. In our experimental assessment of the meaningful impact strategy described in Section VI, we evaluated each strategy's effectiveness at detecting each of five distinct types of faults. These faults represent all of the mutation faults that we considered appropriate for programs that are implementations of this type of specification. We next present an example to illustrate our basic strategy.

Example: Consider the Boolean formula $ab(cd+e)$. Table I lists the test conditions required by the basic meaningful impact strategy for testing an implementation of this formula. Each row of the table describes the test condition to be satisfied by a test case in order to guarantee that a literal occurrence has meaningful impact on the specified outcome. For example, row 1 specifies that for a to have meaningful impact on outcome 1, the test case should satisfy the following condition:

$$(a = 1, b = 1, cd + e = 1).$$

For each such condition, the test set must contain at least one test case that satisfies that condition.

Notice that several test conditions may be satisfied by a single test case. For example, the test conditions in rows 5 and 7 are identical. Also, it is possible to choose a test case that satisfies two or more different test conditions. For example, the test case $(a = 1, b = 1, c = 0, d = 1, e = 1)$ satisfies the conditions of rows $1, 3$, and 9. Hence, the cardinality of a test set that satisfies these test conditions can be less than the number of test conditions themselves.

Notice, too, that the above strategy is not completely deterministic, in the sense that it allows choice among test cases that satisfy a specified condition. Thus, in row 1 of Table I, any assignments to the variables c, d, and e that cause the expression $cd + e$ to evaluate to 1 is acceptable, and no preference among the five such assignments is given.

Foster's strategy resolves this nondeterminism by setting $c = 0$ and $d = 0$ to satisfy $(cd = 0)$ in row 9, because c and d are in "or" relationship with e. Similarly, to enforce $(cd + e = 1)$ in row 1, Foster's rule would set $(c = 1, d = 1, e = 1)$, because they are in "and" relationship with a. One problem with always using this algorithm to resolve the nondeterminism is that it is possible that the test cases required by Foster's rule all evaluate correctly, whereas some other test case that satisfies the test condition evaluates incorrectly and would thereby expose the presence of a fault. For example, if the specification was $ab(cd + e)$, Foster would select the test case $(a = 1, b = 1, c = 1, d = 1, e = 1)$ to satisfy row 1 of Table I. If the implementation was $ab(cd + ce + \bar{d}e)$, then this test case would cause both the specification and the implementation to evaluate to 1. However, there are five distinct test cases that satisfy the test condition. For one of those test cases, $(a = 1, b = 1, c = 0, d = 1, e = 1)$, the specification evaluates to 1, and the implementation evaluates to 0, thereby exposing the presence of the fault. By randomly sampling the set of all test cases that satisfy a given condition, it is possible that a much larger class of faults will be detected.

In addition, note that some test cases determined by using Foster's rule may be infeasible. For example, if variable c represented the condition $(height > 5)$ and variable e represented the condition $(height < 4)$, c and e could not both be 1 in the same test case. In such a situation, the test cases generated by Foster's rule for rows $1, 2, 3$, and 4 would be infeasible, because they would require both c and e to be 1, whereas the assignments $(c = 1,\ d = 1,\ e = 0)$ and $(c = 0,\ d = 0,\ e = 1)$ satisfy the condition $(cd + e = 1)$ and are both feasible. Thus, some test cases that satisfy the requirements of the meaningful impact strategy may be missed if Foster's rule is applied, and it is possible that an infeasible test case may be selected by using the rule, even though a feasible one exists. We have included Foster's algorithm as a seventh way of generating test cases by our tool, and include it in our empirical assessment.

It is interesting to note that when the basic meaningful impact strategy is applied to textually different, but equivalent, formulas, it may yield different sets of test cases. Because our

TABLE II
TEST CONDITIONS REQUIRED FOR THE FORMULA $a_1b_1c_1d_1 + a_2b_2e_1$

Row#	Direct Effect		Test Condition				
	Of Literal	On Outcome	a	b	c	d	e
1	a_1	1	1	1	1	1	0
2	a_1	0	0	1	1	1	ϕ
3	b_1	1	1	1	1	1	0
4	b_1	0	1	0	1	1	ϕ
5	c_1	1	1	1	1	1	0
6	c_1	0	1	1	0	1	0
7	d_1	1	1	1	1	1	0
8	d_1	0	1	1	1	0	0
9	a_2	1	1	1	$cd=0$		1
10	a_2	0	0	1	ϕ	ϕ	1
11	b_2	1	1	1	$cd=0$		1
12	b_2	0	1	0	ϕ	ϕ	1
13	e_1	1	1	1	$cd=0$		1
14	e_1	0	1	1	$cd=0$		0

tool is intended to be used to test *any* implementation of a specification, we want the test sets selected by the tool to be independent of the format of the specification. Therefore, though the specification can be entered in any form, the tool first converts the formula to disjunctive normal form, and uses the resulting formula to generate test cases. Thus, for the formula $ab(cd + e)$ of the above example, the tool would first convert it to $abcd + abe$ and generate test cases for the selected variant of the basic meaningful impact strategy for this disjunctive normal form representation of the formula. Using subscripts to distinguish among literal occurrences, we represent the formula $a_1b_1c_1d_1 + a_2b_2e_1$. The test conditions for the basic meaningful impact strategy for this representation of the formula are shown in Table II. Note that the test set associated with the disjunctive normal form representation is somewhat different than that associated with the other representation shown in Table I.

There was a second reason why we chose to have our tool generate test cases from a disjunctive normal form representation of a formula. As mentioned above, our original motivation for addressing this problem was the availability of a formal specification for TCAS II, a real aircraft collision avoidance system. This specification was written in terms of AND–OR tables, which is simply one way of representing a disjunctive normal form formula. Thus, the specifiers and those reading the specification parts were dealing with that type of representation. In Section VI, we list all of the specifications considered in our evaluation. They are generally not in disjunctive normal form, because when we translated them from the AND–OR table representation to the Boolean formula representation, in many cases, a nondisjunctive normal form representation was most natural. In addition, many of the specifications that we considered contained macros that we expanded. The inclusion of those expansions always led to formulas that were not in disjunctive normal form.

The fact that our strategy and tool work on a disjunctive normal form representation of the specification, which is not necessarily the form that the specification was written in, means that even though the strategy was expressly designed to detect Variable Negation Faults, it may not detect all such faults in the original version of the specification. Our empirical results in Section VI indicate, however, that the strategies are, in practice, extremely successful at detecting them (and most of the other faults considered).

In the above example, we illustrated the basic strategy in terms of test conditions. We now present the strategy by using the notation introduced in Section II. When testing a Boolean formula in irreducible disjunctive normal form, the following test cases are required by the basic meaningful impact strategy.

1) Select one test point from each nonempty U_i of F. A point selected from U_i demonstrates the meaningful impact of each of the literals in product-term p_i on the 1 outcome, because all product-terms other than p_i will evaluate to 0 for that point.
2) Select one test point from each $N_{i,j}$ of F. A point picked from $N_{i,j}$ demonstrates the meaningful impact of the literal $l_{i,j}$ on the 0 outcome, because, for such a point, all other literals in the product-term p_i evaluate to 1, and all product-terms other than p_i evaluate to 0.

The definition of the sets U_i and $N_{i,j}$ define *test conditions* that must be satisfied. Although our automatic test case generation tool forms the sets U_i and $N_{i,j}$, and, when necessary, O and M, and although it selects points from these formally defined sets, when human testers use this approach, it is frequently easier to work in terms of test conditions. In fact, our original intuition was based on this perspective. To illustrate the relationship between the two forms of the definitions for selecting test cases, we present in Fig. 1 the derived test conditions and all U_i's and $N_{i,j}$'s for the formula.

IV. THE FAULT DETECTION ABILITY OF THE BASIC STRATEGY

In this section, we analyze the fault detection ability of the basic meaningful impact strategy when applied to a disjunctive normal form representation of a given Boolean formula. We also discuss the fault detection ability of the strategy when applied to the canonical disjunctive normal form of a given Boolean formula. For that case, the strategy is completely deterministic.

A. Faults Guaranteed to Be Detected

The following categories of incorrect implementations G of a specification F are *guaranteed* to be detected by the basic meaningful impact testing strategy, and hence all of the variants presented in Section V will also detect these types of faults.

1) An implementation G that evaluates to 0 for every point in at least one nonempty U_i is guaranteed to be recognized as faulty, because each set of unique true points is sampled at least once. If no point in that set evaluates to 1, then any point from that set would cause the implementation to evaluate to 0 while the specification indicates that it should evaluate to 1, and hence the fault would be detected. For example, any test set that satisfies test conditions C_1 through C_9 in Fig. 1 for testing the formula $F = ac + a\bar{b} + \bar{c}d$, will detect the incorrect implementation $G = a\bar{b} + \bar{c}d$, because G is false for every point of the unique true set associated with the term ac.
2) An implementation G that evaluates to 1 for every point in some nonempty $N_{i,j}$ is guaranteed to be recognized

Specification of Formula $ac + a\bar{b} + \bar{c}d$ Required Test Conditions for True Points:

Product Term	Unique True Points Represented by	Required Test Conditions Cond #	a	b	c	d
p_1 : ac	U_1: $abc = 14, 15$	C_1	1	1	1	ϕ
p_2 : $a\bar{b}$	U_2: $a\bar{b}\bar{c}\bar{d} = 8$	C_2	1	0	0	0
p_3 : $\bar{c}d$	U_3: $(\bar{a} + b)\bar{c}d = 1, 5, 13$	C_3	$a\bar{b} = 0$		0	1

Required Test Conditions for False Points:

Product Term	$D_{i,j}$ Represented by		$N_{i,j}$ Represented by		Required Test Conditions Cond #	a	b	c	d
p_1 : ac	$D_{1,1}$:	$\bar{a}c$=2,3,6,7	$N_{1,1}$:	$\bar{a}c$=2,3,6,7	C_4	0	ϕ	1	ϕ
	$D_{1,2}$:	$a\bar{c}$=8,9,12,13	$N_{1,2}$:	$ab\bar{c}\bar{d}$=12	C_5	1	1	0	0
p_2 : $a\bar{b}$	$D_{2,1}$:	$\bar{a}\bar{b}$=0,1,2,3	$N_{2,1}$:	$\bar{a}\bar{b}(c + \bar{d})$=0,2,3	C_6	0	0	$\bar{c}d = 0$	
	$D_{2,2}$:	ab=12,13,14,15	$N_{2,2}$:	$ab\bar{c}\bar{d}$=12	C_7	1	1	0	0
p_3 : $\bar{c}d$	$D_{3,1}$:	cd=3,7,11,15	$N_{3,1}$:	$\bar{a}cd$=3,7	C_8	0	ϕ	1	1
	$D_{3,2}$:	$\bar{c}\bar{d}$=0,4,8,12	$N_{3,2}$:	$(\bar{a} + b)\bar{c}\bar{d}$=0,4,12	C_9	$a\bar{b} = 0$		0	0

Example Test Set:

Test Case #	a	b	c	d	Decimal Rep	Test Conditions Satisfied
1	1	1	1	0	14	C_1
2	1	0	0	0	8	C_2
3	0	1	0	1	5	C_3
4	0	0	1	0	2	C_4, C_6
5	1	1	0	0	12	C_5, C_7, C_9
6	0	0	0	0	0	C_9, C_6
7	0	0	1	1	3	C_8, C_4, C_6

Fig. 1. Example.

as faulty, because each $N_{i,j}$ contains points that should cause F to evaluate to 0, and each $N_{i,j}$ is sampled at least once. Whatever point of $N_{i,j}$ is selected, G will evaluate to 1, but F specifies that it should evaluate to 0. For example, any test set that satisfies test conditions C_1 through C_9 in Fig. 1 for testing the specification $F = ac + a\bar{b} + \bar{c}d$ will detect the incorrect implementation $G = ac + a\bar{b} + \bar{c}d + \bar{a}c$, because $N_{1,1} = \bar{a}c$, and this is the extra product-term of G. Hence, this fault would be guaranteed to be detected.

B. Faults That Will Go Undetected

Any faulty implementation G of a specification F that satisfies *all three* of the following conditions will not be detected by the meaningful impact strategy.

1) All of the unique true points associated with product-terms of F cause G to evaluate to 1.
2) All near false points, N of F cause G to evaluate to 0.
3) Either at least one point in the set of remaining false points of F causes G to evaluate to 1, or at least one overlapping true point of F causes G to evaluate to 0.

Conditions 1 and 2 guarantee that all of the sampled points are correct, whereas condition 3 guarantees that some unsampled set contains a point that fails. The basic strategy does not sample either of the sets M or O. In Section V, we introduce two enhanced strategies that do sample these sets. For example, consider the specification $S = a\bar{b}$ and an implementation $I = a\bar{b} + \bar{a}b$. The additional term $\bar{a}b$ in the implementation evaluates to 0 for the only true point of S, and for the two near false points of S. Therefore, a test set generated by using the meaningful impact strategy would not detect the incorrect implementation I. If, however, the situation were reversed, with $S = a\bar{b} + \bar{a}b$ and $I = a\bar{b}$, the fault would be detected by the strategy, because the unique true point $(a = 0, b = 1)$ for the second term of S must be selected, and would incorrectly evaluate to 0 in the implementation.

C. Faults That May or May Not Be Detected

The following categories of incorrect implementations of the formula F *may or may not be* detected by the meaningful impact strategy.

1) An implementation G that evaluates to 1 for some, but not all, unique true points for a term of F. Since points are randomly selected from each U_i, the test generation strategy may or may not select a point that exposes the fault. For example, consider the test set used in Fig. 1 for testing $F = ac + a\bar{b} + \bar{c}d$. This test set will detect an incorrect implementation $G = abcd + a\bar{b} + \bar{c}d$, because test case 1 evaluates to 0 instead of 1. If test case 1 were instead $(a = 1, b = 1, c = 1, d = 1)$, it would still satisfy test condition C_1, but would fail to detect the fault. This is because this test case evaluates to 1 as specified.
2) An implementation G that evaluates to 1 for some, but not all, near false points for a given term of F. Again, the problem is that because each $N_{i,j}$ is randomly sampled, the test generation strategy may or may not select a point that exposes the fault. For example, consider again the test set used in Fig. 1 for testing $F = ac + a\bar{b} + \bar{c}d$.

This test set will not detect the incorrect implementation $G = ac + a\bar{b} + \bar{c}d + \bar{a}bcd$. Note that $N_{3,1}$ of F contains two points: 0011 and 0111. Test case 7 ($a = 0, b = 0, c = 1, d = 1$) happens to evaluate to 0 for G, and thus does not expose the fault. If test case 7 were instead ($a = 0, b = 1, c = 1, d = 1$), it would still satisfy test condition C_8 and would successfully detect the fault.

D. Discussion

A little reflection will help to illuminate these categories. Essentially, they state that if the subdomains that are created by dividing the domain into the subsets described in Section II contain only points that evaluate correctly, but if there are points in subsets that will not be sampled that evaluate incorrectly, faults will definitely go undetected. If, on the other hand, every element of a sampled subdomain causes a failure, the associated fault is guaranteed to be exposed. If the sampled subdomains contain both failure-causing inputs and points that evaluate correctly, then the likelihood of detecting a fault depends on the distribution of points in that subdomain. We next show that if the test cases are generated by using the canonical disjunctive normal form representation of formula F, no probability analysis is required, because all faults are either definitely detected or definitely missed.

E. Special Case: Canonical Disjunctive Normal Form

Recall that in the canonical disjunctive normal form of a Boolean formula, all product-terms are minterms. If a product-term p_i in the sum-of-products form of a formula is a minterm, its associated set of unique true points U_i and sets $D_{i,j}$ each contain only a single point. Therefore, each $N_{i,j}$ contains at most one point, because it is a subset of $D_{i,j}$. As mentioned above, because our basic test generation strategy requires the selection of one point from each U_i and one point from each nonempty $N_{i,j}$, the strategy is completely deterministic. For this special case, therefore, the fault detection ability of the test strategy is easy to describe.

The following categories of incorrect implementations of the formula F are *guaranteed* to be detected.

1) An implementation G that evaluates to 0 for some point x in the true set of F will be recognized as faulty, because the specification indicates that it should evaluate to 1.
2) An implementation G that evaluates to 1 for some near false point of F will be recognized as faulty, because the specification indicates that it should evaluate to 0.

The following category of incorrect implementations of the formula F *will not be* detected.

An implementation G that evaluates to

1) 1 for all true points of F,
2) and 0 for all near false points of F,
3) and 1 for at least one false point of F.

Note that because the strategy is completely deterministic, there can never be a situation in which it is uncertain whether a fault will be detected. The fault detection ability of the test set generated by using the canonical disjunctive normal form is superior to that of test sets generated by using any other irreducible sum-of-products representation. Of course, this superior fault detection ability comes as a result of additional test cases. In general, the number of test cases generated by using the canonical disjunctive normal form of a formula will be greater than or equal to that using any other sum-of-products representation of that formula.

V. Enhancing the Basic Strategy

With the above analysis in mind, we now introduce a family of algorithms for automatically generating test cases for implementations intended to satisfy specifications that are Boolean formulas. Four of the strategies enhance the basic meaningful impact strategy by increasing the number of test cases selected from each set, and two of these strategies also sample sets not sampled in the basic meaningful impact strategy. In Section VI, we discuss our experience using these strategies.

Of course, as the number of test cases required from each such set increases, the total number of test cases required by the strategy increases. Our empirical results presented in Section VI show this tradeoff in terms of effectiveness gains relative to cost.

In each case, when selecting points from a set, the selection is done randomly by using a uniform distribution. In addition, once a point is selected, it is removed from consideration from all other sets. The first two variants essentially implement the basic meaningful impact strategy. ONE is a straightforward implementation of the strategy, and MIN attempts to optimize the ONE strategy.

MIN: One point is selected from the set of unique true points associated with each term, and the minimum set of points needed to satisfy the basic meaningful impact strategy are selected from the set of near false points for the formula.

ONE: One point is selected from the set of unique true points associated with each term, and one point is selected from each set of near false points $N_{i,j}$.

In the following variants of the meaningful impact strategy, more than one point is selected from a given set. In most cases, the number of points selected is determined by the size of the set being sampled.

MANY-A: For a set of 2^X unique true points associated with a term, $\lceil X \rceil$ points are selected from the set, and $\lceil X \rceil$ points are selected from each set of near false points, $N_{i,j}$, of size 2^X. If $X = 0$ for some set, then one point is selected from that set.

MANY-B: For a set of 2^X unique true points associated with a term, $\lceil X \rceil$ points are selected from the set, and $\lceil X \rceil$ points are selected from each set of near false points, $N_{i,j}$ of size 2^X. In addition, $\lceil X \rceil$ points are selected from the set of overlapping true points of size 2^X, and $\lceil X \rceil$ points are selected from the set of remaining false points of size 2^X. If $X = 0$ for some set, then one point is selected from that set.

MAX-A: Every point of the set of unique true points associated with each term is selected, and every point from each set of near false points, $N_{i,j}$, is selected.

1. $\overline{(ab)}(d\bar{e}\,\bar{f} + \bar{d}e\,\bar{f} + \bar{d}\,\bar{e}\,\bar{f})(ac(d+e)h + a(d+e)\bar{h} + b(e+f))$
2. $(a((c+d+e)g + af + c(f+g+h+i)) + (a+b)(c+d+e)i)\ \overline{(ab)}\ \overline{(cd)}\ \overline{(ce)}\ \overline{(de)}\ \overline{(fg)}\ \overline{(fh)}\ \overline{(fi)}\ \overline{(gh)}\ \overline{(hi)}$
3. $(a(\bar{d} + \bar{e} + de\overline{(\bar{f}\,ghi + \bar{g}\,hi)}\ \overline{(\bar{f}\,glk + \bar{g}\,ik)}) + \overline{(\bar{f}\,ghi + \bar{g}\,hi)}\ \overline{(\bar{f}\,glk + \bar{g}\,ik)}(b + c\bar{m} + f))\ (a\,\bar{b}\,\bar{c} + \bar{a}\,b\bar{c} + \bar{a}\,\bar{b}c)$
4. $a(\bar{b} + \bar{c})d + e$
5. $a(\bar{b} + \bar{c} + bc\overline{(\bar{f}\,ghi + \bar{g}\,hi)}\ \overline{(\bar{f}\,glk + \bar{g}\,ik)}) + f$
6. $(\bar{a}b + a\bar{b})\ \overline{(cd)}\ \overline{(f\,\bar{g}\,\bar{h} + \bar{f}\,g\bar{h} + \bar{f}\,\bar{g}\,\bar{h})}\ \overline{(jk)}((ac + bd)e(f + (i(gj + hk))))$
7. $(\bar{a}b + a\bar{b})\ \overline{(cd)}\ \overline{(gh)}\ \overline{(jk)}((ac + bd)e(\bar{i} + \bar{g}\,\bar{k} + \bar{j}(\bar{h} + \bar{k})))$
8. $(\bar{a}b + a\bar{b})\ \overline{(cd)}\ \overline{(gh)}((ac + bd)e(fg + \bar{f}\,h))$
9. $\overline{(cd)}\ (\bar{e}\,f\,\bar{g}\,\bar{a}(bc + \bar{b}d))\ \overline{(jk + \bar{j}\,l + m)}$
10. $a\,\bar{b}\,\bar{c}\,d\bar{e}\,f(g + \bar{g}(h+i))\ \overline{(jk + \bar{j}\,l + m)}$
11. $a\,\bar{b}\,\bar{c}(\overline{(f(g + \bar{g}(h+i)))} + f(g + \bar{g}(h+i))\,\bar{d}\,\bar{e})\ \overline{(jk + \bar{j}\,l\,\bar{m})}$
12. $a\,\bar{b}\,\bar{c}(f(g + \bar{g}(h+i))(\bar{e}\,\bar{n} + d) + \bar{n}(jk + \bar{j}\,l\,\bar{m})$
13. $a + b + c + \bar{c}\,\bar{d}\,e\,f\,\bar{g}\,\bar{h} + i(j + k)\bar{l}$
14. $ac(d+e)h + a(d+e)\bar{h} + b(e+f)$
15. $a((c+d+e)g + af + c(f+g+h+i)) + (a+b)(c+d+e)i$
16. $a(\bar{d} + \bar{e} + de\overline{(\bar{f}\,ghi + \bar{g}\,hi)}\ \overline{(\bar{f}\,glk + \bar{g}\,ik)}) + \overline{(\bar{f}\,ghi + \bar{g}\,hi)}\ \overline{(\bar{f}\,glk + \bar{g}\,ik)}(b + c\bar{m} + f)$
17. $(ac + bd)e(f + (i(gj + hk)))$
18. $(ac + bd)e(\bar{i} + \bar{g}\,\bar{k} + \bar{j}(\bar{h} + \bar{k}))$
19. $(ac + bd)e(fg + \bar{f}\,h)$
20. $\bar{e}\,f\,\bar{g}\,\bar{a}(bc + \bar{b}d)$

Fig. 2. Specifications.

MAX-B: Every point of the set of unique true points associated with each term is selected, and every point from each set of near false points, $N_{i,j}$, is selected. In addition, $\lceil X \rceil$ points are selected from the set of overlapping true points of size 2^X, and $\lceil X \rceil$ points are selected from the set of remaining false points of size 2^X. If $X = 0$ for some set, then one point is selected from that set.

VI. Empirical Results

In this section, we consider the cost and effectiveness of using the meaningful impact strategies. We selected 13 of the larger transition specifications from the TCAS II specification. They varied in size from 5 to 14 variables, with the average containing 10 distinct variables. For each of these specifications, we checked to see whether there were any dependencies among variables. For example, if a variable X represented the altitude of an airplane being within a certain range, and a different variable Y represented the fact that the airplane was in some different, disjoint altitude range, then it would be impossible for both of the variables to be true simultaneously. A clause $(\overline{XY})$ reflecting this fact would therefore be added to the formula. Whenever such variable dependencies existed, we represented the specifications in two ways: first adding clauses to reflect these dependencies, and then ignoring the dependencies. In seven of the specifications, we identified variable dependencies, and thus had a set of 20 specifications to work with. The 20 specifications are listed in Fig. 2.

Our tool automatically generated a test set to satisfy the selected strategy for each of the 20 Boolean specifications and for each of the seven strategies supported by our tool (six variants of the basic meaningful impact strategy plus Foster's strategy). Table III shows the size of the generated test set, represented as a percentage of the size of the exhaustive test set for that specification. For example, specification 1 contains seven variables, and hence exhaustive testing would require $2^7 = 128$ test cases. Therefore, because the MIN strategy generated 26 test cases for this formula, the entry in Table III for this strategy and specification is $\frac{26}{128} \times 100 = 20.3$.

TABLE III
Percentage of Exhaustive Test Set Generated

Spec No.	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B	No. of Vars
1	20.3	23.4	23.4	26.6	29.7	39.1	45.3	7
2	11.7	18.8	18.8	18.8	20.1	22.7	25.6	9
3	1.3	5.4	5.4	23.4	23.8	62.0	62.5	12
4	18.8	28.1	28.1	46.9	53.1	87.5	96.9	5
5	3.3	6.4	6.4	14.5	16.0	69.7	87.3	9
6	2.2	2.9	2.9	4.6	4.8	6.3	6.8	11
7	2.5	6.4	6.4	13.6	14.3	20.3	22.1	10
8	12.5	22.3	22.3	31.3	33.2	43.8	49.2	8
9	12.5	12.5	12.5	12.5	15.6	12.5	20.3	7
10	0.5	1.1	1.1	2.1	2.2	2.9	3.2	13
11	0.6	2.2	2.2	7.7	8.0	22.9	25.3	13
12	0.1	0.4	0.4	2.4	2.5	24.7	25.6	14
13	0.3	0.5	0.5	3.0	3.3	41.5	41.8	12
14	8.6	17.2	17.2	48.4	54.7	71.1	89.1	7
15	2.7	8.4	8.4	28.9	31.1	58.0	93.0	9
16	0.8	3.1	3.1	15.6	16.1	47.8	48.5	12
17	0.6	1.9	1.9	9.5	10.1	46.2	49.6	11
18	1.3	4.5	4.5	19.0	20.3	48.2	56.6	10
19	6.3	18.8	18.8	46.5	49.6	67.2	89.1	8
20	9.4	10.9	10.9	10.9	13.3	17.8	26.6	7
avg	5.8	9.8	9.8	19.3	21.1	40.6	48.2	10

Similarly, the entry for specification 12 for the MIN strategy is $\frac{21}{16\,384} \times 100 = 0.1$. The other entries are computed similarly. An average ratio for each strategy is also shown in the table. These ranged from an average of 5.8% for the MIN strategy, to an average of 48.2% for MAX-B. We argue, in this case, that it is meaningful to assess the cost solely on the basis of the number of test cases generated, because all test case generation was done automatically. In general, the more test cases generated, the more time it takes to generate them and run them.

We also examined the effectiveness of each of the strategies as assessed in terms of their mutation scores. We considered five mutation operators, all those we considered applicable to a Boolean formula. The following are the five operators.

1) *Variable Negation Fault (VNF):* Replace one occurrence of a variable by its negation.
2) *Expression Negation Fault (ENF):* Replace an expression by its negation.
3) *Variable Reference Fault (VRF):* Replace one occurrence of a variable by another, or by a constant.

TABLE IV
MUTATION SCORES FOR MEANINGFUL IMPACT STRATEGIES

Spec. No.	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
1	100.0	100.0	100.0	100.0	100.0	100.0	100.0
2	99.8	99.8	99.8	99.8	100.0	99.8	100.0
3	97.8	100.0	100.0	100.0	100.0	100.0	100.0
4	100.0	100.0	100.0	100.0	100.0	100.0	100.0
5	98.4	100.0	100.0	100.0	100.0	100.0	100.0
6	92.7	94.9	94.9	92.7	95.4	95.4	95.4
7	97.6	97.6	97.6	97.9	97.9	97.9	97.9
8	100.0	100.0	100.0	100.0	100.0	100.0	100.0
9	100.0	100.0	100.0	100.0	100.0	100.0	100.0
10	100.0	100.0	100.0	100.0	100.0	100.0	100.0
11	98.2	99.4	99.4	100.0	100.0	100.0	100.0
12	96.5	97.7	97.7	100.0	100.0	100.0	100.0
13	95.0	97.5	97.5	100.0	100.0	100.0	100.0
14	94.7	97.7	97.7	100.0	100.0	100.0	100.0
15	94.6	95.0	95.0	100.0	100.0	100.0	100.0
16	92.8	99.5	99.5	100.0	100.0	100.0	100.0
17	99.4	99.4	99.4	100.0	100.0	100.0	100.0
18	100.0	100.0	100.0	100.0	100.0	100.0	100.0
19	100.0	100.0	100.0	100.0	100.0	100.0	100.0
20	100.0	100.0	100.0	100.0	100.0	100.0	100.0
avg	97.9	98.9	98.9	99.5	99.7	99.7	99.7

TABLE V
MUTATION SCORES FOR SIZE-EQUIVALENT RANDOM TEST SETS

Spec. No.	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
1	86.2	86.6	86.6	98.9	98.9	100.0	100.0
2	30.7	48.7	48.7	48.7	59.2	59.6	59.6
3	54.4	70.8	70.8	95.1	95.1	98.2	98.2
4	46.7	55.6	55.6	64.4	64.4	91.1	100.0
5	33.6	48.8	48.8	72.3	73.0	97.3	97.3
6	12.1	12.1	12.1	14.2	22.3	25.3	25.3
7	30.6	58.1	58.1	83.8	84.2	84.2	84.2
8	76.8	96.7	96.7	98.6	98.6	99.1	99.5
9	94.5	94.5	94.5	94.5	94.5	94.5	94.5
10	3.6	9.1	9.1	16.2	16.2	24.5	24.5
11	28.9	63.4	63.4	94.2	94.2	96.0	96.0
12	15.1	36.7	36.7	84.6	84.6	100.0	100.0
13	30.7	31.2	31.2	82.7	82.7	100.0	100.0
14	47.7	75.8	75.8	94.7	100.0	100.0	100.0
15	24.8	75.7	75.7	97.3	97.3	100.0	100.0
16	39.7	67.3	67.3	98.3	98.3	98.3	98.3
17	26.3	71.3	71.3	95.3	95.3	100.0	100.0
18	26.6	70.3	70.3	96.2	96.2	100.0	100.0
19	87.5	96.4	96.4	100.0	100.0	100.0	100.0
20	57.3	57.3	57.3	57.3	58.4	65.2	65.2
avg	42.7	61.3	61.3	79.4	80.7	86.7	87.1

4) *Operator Reference Fault (ORF):* Replace one Boolean operator with another.
5) *Associative Shift Fault (ASF):* Change the associativity of terms. Thus, for example, replace $A(B + C)$ with $(AB) + C$.

Table IV shows the mutation score for each strategy and each specification, as well as the average mutation score for each strategy. In each case, all mutants of type VNF, ENF, VRF, ORF, and ASF were included in the analysis. The lowest mutation score encountered for any strategy and any specification was 92.7. The averages ranged from a low of 97.9 for the MIN strategy to a high of 99.7. Thus, on average, all of the meaningful impact strategy variants did extremely well.

Because these results were so encouraging, we decided to compare the mutation scores obtained by the various meaningful impact strategies to the mutation scores obtained by randomly selected test sets of exactly the same size. In Table V, we present the results for randomly generated test sets. The columns are labeled with meaningful impact strategy variant names in order to indicate the size of the test set. Thus, the column labeled MIN in Table V shows the mutation score of a randomly generated test set of size equal to that of the set generated by our tool to satisfy the MIN strategy.

TABLE VI
AVERAGES

	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
Mut Scores - D.E. T.S.	97.9	98.9	98.9	99.5	99.7	99.7	99.7
Mut Scores - Ran T.S.	42.7	61.3	61.3	79.4	80.7	86.7	87.1
Pctg Ex T.S.	5.8	9.8	9.8	19.3	21.1	40.6	48.2

TABLE VII
PERCENTAGES OF EXHAUSTIVE TEST SET GENERATED BY RANGES

No. of Vars	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
5-8	12.6	19.0	19.0	31.9	35.6	48.5	59.5
9-11	3.5	7.0	7.0	15.5	16.7	38.8	48.7
12-14	0.6	2.1	2.1	9.0	9.3	33.6	34.5

TABLE VIII
MUTATION SCORES BY TEST STRATEGY FOR VARIABLE NEGATION FAULTS (VNF)

Spec. No.	Ineq Muts	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
6	25	92.0	96.0	96.0	92.0	96.0	96.0	96.0
15	18	94.4	94.4	94.4	100.0	100.0	100.0	100.0
others	5-46	100.0	100.0	100.0	100.0	100.0	100.0	100.0
avg		99.3	99.5	99.5	99.6	99.8	99.8	99.8

TABLE IX
MUTATION SCORES BY TEST STRATEGY FOR EXPRESSION NEGATION FAULTS (ENF)

Spec. No.	Ineq Muts	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
all	4-45	100.0	100.0	100.0	100.0	100.0	100.0	100.0

TABLE X
MUTATION SCORES BY TEST STRATEGY FOR VARIABLE REFERENCE FAULTS (VRF)

Spec. No.	Ineq Muts	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
3	584	97.3	100.0	100.0	100.0	100.0	100.0	100.0
5	191	97.9	100.0	100.0	100.0	100.0	100.0	100.0
6	281	91.8	94.0	94.0	91.8	94.7	94.7	94.7
7	221	96.8	96.8	96.8	97.3	97.3	97.3	97.3
11	261	98.1	99.6	99.6	100.0	100.0	100.0	100.0
12	252	96.0	97.6	97.6	100.0	100.0	100.0	100.0
13	163	93.9	96.9	96.9	100.0	100.0	100.0	100.0
14	93	92.5	96.8	96.8	100.0	100.0	100.0	100.0
15	162	93.8	94.4	94.4	100.0	100.0	100.0	100.0
16	472	90.9	99.4	99.4	100.0	100.0	100.0	100.0
17	132	99.2	99.2	99.2	100.0	100.0	100.0	100.0
others	30-472	100.0	100.0	100.0	100.0	100.0	100.0	100.0
avg		97.4	98.7	98.7	99.5	99.6	99.6	99.6

Table VI directly compares the average mutation scores of each variant of the meaningful impact strategy to the average scores of random test sets of equal size. The table's third row shows the average size of the meaningful impact and random test sets as a percentage of the size of the exhaustive test set. In all cases, the meaningful impact strategies, on average, performed significantly better than random testing, with the greatest benefit realized for the less demanding variants. This is as expected, because the less demanding strategies required the smallest test sets.

In general, the more variables there are in a specification, the smaller is the ratio of the meaningful impact test set size to the exhaustive test set size. Table VII shows the average ratio for each strategy for three groupings of specifications based on the number of variables in the specification.

Tables VIII through XII show the mutation score, by mutation type, for each specification and each strategy. In most cases, the mutation score for all strategies was 100, and

TABLE XI
MUTATION SCORES BY TEST STRATEGY FOR OPERATOR REFERENCE FAULTS (ORF)

Spec. No.	Ineq Muts	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
6	72	92.6	96.3	96.3	92.6	96.3	96.3	96.3
others	12-121	100.0	100.0	100.0	100.0	100.0	100.0	100.0
avg		99.6	99.8	99.8	99.6	99.8	99.8	99.8

TABLE XII
MUTATION SCORES BY TEST STRATEGY FOR ASSOCIATIVE SHIFT FAULTS (ASF)

Spec. No.	Ineq Muts	MIN	ONE	FOSTER	MANY-A	MANY-B	MAX-A	MAX-B
2	9	88.9	88.9	88.9	88.9	100.0	88.9	100.0
11	9	88.9	88.9	88.9	100.0	100.0	100.0	100.0
12	10	90.0	90.0	90.0	100.0	100.0	100.0	100.0
15	8	87.5	87.5	87.5	100.0	100.0	100.0	100.0
others	2-19	100.0	100.0	100.0	100.0	100.0	100.0	100.0
avg		97.8	97.8	97.8	99.4	100.0	99.4	100.0

these are listed as "others" or "all" in the tables. We include the number of inequivalent mutants of each type for each specification. When a number of specifications have been grouped together because all of their mutation scores were 100, we indicate the range of the number of inequivalent mutants for the specifications in that group. We also show the average mutation score for each strategy for each of the five fault types.

It was not surprising that all of the strategies were very good at detecting Variable Negation Faults (VNF), because the basic strategy was designed explicitly to detect this type of fault. In fact, the reason why they did not detect *all* such faults is that our tool converts the specification into disjunctive normal form and applies the algorithms to that form, rather than the original input version. The mutation analyzer, in contrast, is applied to the originally input form, and thus there may be minor differences between the VNF's of the two forms. The success of the strategies at detecting the other faults is more interesting, and extremely encouraging.

Each of the considered mutation operators represented one simple typical fault that might plausibly be introduced during implementation. We hypothesized that those types of faults were in fact relatively difficult to detect, because relatively few points would typically be affected by such a fault. We therefore also generated 500 random Boolean formulas containing exactly the same variables as each of the specifications considered above. In every case, each of the meaningful impact strategy variants obtained a fault detection rate of 100%. That is, the ratio of the number of these randomly generated Boolean formulas that were distinguished from the specification to the total number of inequivalent generated formulas was always 1. As mentioned in Section III, this did not surprise us, because such a randomly generated formula would be very likely to differ substantially from the specification, and the more places that two functions differ, the easier it generally is to detect the presence of these faults.

VII. CONCLUSION

We have presented a formal description of the meaningful impact strategy for testing implementations of Boolean formulas and have analyzed the fault detection ability for both the disjunctive normal form and canonical disjunctive normal form representations of a Boolean formula. We also examined the effectiveness of our strategies empirically, using a set of twenty specifications taken from the specification for a real aircraft collision avoidance system (TCAS II). We used mutation analysis for this portion of the assessment. Selected mutation operators represented all relevant "simple faults" that we identified. Assessed in this way, all of the meaningful impact strategies did very well, ranging from a low average mutation score of 97.9 for MIN to a high average mutation score of 99.7.

To evaluate the effect of the test set size on these results, we also used random selection to generate test sets of exactly the same size as those generated by each of the meaningful impact strategies for each of the specifications, and computed the mutation scores for these random test sets. They ranged from the low average of 42.7 for the random test sets of size equal to MIN test sets, to a high average of 87.1 for the random test sets of size equal to MAX-B test sets. Thus, we saw significant benefit in using any of the variants of the meaningful impact strategy as compared to random testing.

We also examined the cost of the criteria assessed in terms of the number of test cases required by each strategy, and compared these suites to exhaustive test sets. The MIN strategy required the smallest number of test cases, averaging 5.8% of the size of an exhaustive test set. The most costly strategy was MAX-B, which required, on average, test sets that were 48.2% of the size of an exhaustive test set. For larger formulas, these percentages were even smaller. For example, in our sample, formulas containing 12 to 14 variables averaged only 0.6% of an exhaustive test set for the MIN strategy, whereas MAX-B test sets averaged 34.5%. FOSTER is one deterministic instance of the ONE strategy, and had the same performance as ONE.

We concluded that all of the meaningful impact strategies were extremely effective. The MANY-A and MANY-B strategies represented particularly attractive choices for safety-critical systems like TCAS II, because they are both relatively cheap and highly effective. The average test set sizes were, respectively, 19% and 21% of an exhaustive test set, with average detection rates of 99.5% and 99.7%, respectively. For the largest formulas we considered (containing 13 and 14 variables), the average test set size was only 4% of an exhaustive test set, whereas the effectiveness was 100% for both MANY-A and MANY-B. The decrease in the relative size of the test set compared to the size of an exhaustive test set as the number of variables grows is particularly important for large formulas. An exhaustive test set for a formula containing 13 variables contains 8192 points, but 4% of that is 328, a manageable number, particularly because these test cases are automatically generated. An exhaustive test set for a formulas containing 14 variables contains 16 384 points, but 4% of that is 655.

We look forward to continuing our experimentation with these strategies. Our preliminary results indicate that this is worth pursuing. We expect to symbolize additional portions of the TCAS II specification and investigate the cost and effectiveness results for a much larger set of formulas. We are also interested in finding other specifications that can be represented as Boolean formulas. We are currently looking for such examples among specifications for commercial transaction processing software, avionics software, and telecommunications software.

REFERENCES

[1] J.J. Chilenski and S.P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Eng. J.*, submitted for publication.

[2] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Comput.*, Apr. 1978, pp. 34–41.

[3] K.A. Foster, "Sensitive test data for logic expressions," *ACM SIGSOFT Software Eng. Notes*, vol. 9, no. 2, pp. 120–126, Apr. 1984.

[4] P.G. Frankl and E.J. Weyuker, "A formal analysis of the fault-detecting ability of testing methods," *IEEE Trans. Software Eng.*, vol. 19, pp. 202–213, Mar. 1993.

[5] D. Gelperin, "Partial testing of complex decision logic," in preparation.

[6] R. Hamlet, "Theoretical comparison of testing methods," in *Proc. 3rd Symp. Testing, Analysis, and Verification*, 1989, pp. 28–37.

[7] Z. Kohavi, *Switching and Finite Automata Theory*, 2nd ed. New York: McGraw-Hill, 1978.

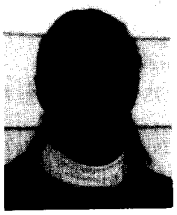
[8] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements specification for process-control systems," Tech. Rep. 92–106, Dept. of Inform. and Comput. Sci., Univ. of Cal., Irvine, Nov. 1992.

[9] K. C. Tai, "Condition-based software testing strategies," *Proc. Compsac 90, 14th Ann. Int. Comput. Software & Applic. Conf.*, Oct. 1990, pp. 564–569.

[10] S. N. Weiss, "What to Compare When Comparing Test Data Adequacy Criteria," *Software Eng. Notes*, vol. 14, pp. 42–49, Oct. 1989.

[11] E.J. Weyuker, "Can we measure software testing effectiveness," *Proc. IEEE-CS Int. Software Metrics Symp.*, May 1993, pp. 100–107.

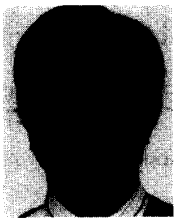
[12] E.J. Weyuker, S.N. Weiss, and R. Hamlet, "Comparison of program testing strategies," in *Proc. 4th Symp. Software Testing, Analysis and Verification (TAV4)*, 1991, pp. 1–10.



E. Weyuker received the M.S.E. degree from the Moore School of Electrical Engineering, University of Pennsylvania, and the Ph.D. degree in computer science from Rutgers University.

She is currently a member of the technical staff in the Software and Systems Research Center at AT&T Bell Laboratories at Murray Hill, NJ, and a Professor of Computer Science at the Courant Institute of Mathematical Sciences of New York University (NYU), New York, NY, where she has been on the faculty since 1977. Before coming to NYU, she was on the faculty of the City University of New York, was a systems engineer for IBM, and was a programmer for Texaco, Inc. Her research interests are in software engineering, particularly software testing and reliability, and software complexity measures, and has published many papers in those areas. She is also interested in the theory of computation, and is the author of a book (with Martin Davis and Ron Sigal), *Computability, Complexity, and Languages*, published by Academic Press. She spends a large portion of her nonworking time working with elementary school–age children. She is particularly eager to encourage young girls to become interested in science, and, for this reason, is a volunteer scientist for the Science by Mail program. She also regularly teaches hands-on science to first graders in a local public school.

She is currently a member of the Executive Committee of the IEEE Computer Society Technical Committee on Software Engineering, and is a member of the Editorial Board of *ACM Transactions on Software Engineering and Methodology* (TOSEM). She is also a member of the ACM Committee on the Status of Women and Minorities, and a former member of the CRA Committee on the Status of Women. She was Secretary/Treasurer of ACM SIGSOFT and has been an ACM National Lecturer.



T. Goradia received the B.Tech. degree in computer science from Indian Institute of Technology, Bombay, in 1984, the M.S. degree in computer science from the University of Wisconsin at Madison in 1985, and the Ph.D. degree in computer science from New York University in 1993.

He was a research engineer at Honeywell Corporate Systems Development Division in Minneapolis from 1986 to 1989. Since 1991, he has been a member of technical staff at Siemens Coporate Research, Inc. in Princeton, NJ.

His research interests include program and execution analyses, software testing and debugging, and design for testability.

A. Singh received the B.E. degree (with honors) from Birla Institute of Technology and Science, Pilani, India, in 1982.

He is currently working on the M.S. degree in computer science at New York University, New York, NY. Since 1991, he has been with Investment Support Systems, Inc, where his interests are object-oriented programming languages and OODBS.