Empirical Studies of Predicate-Based Software Testing*

M. A. Vouk, K. C. Tai, and A. Paradkar
Computer Science Department, Box 8206
North Carolina State University Raleigh, NC 27659-8206, USA
{vouk, kct}@adm.csc.ncsu.edu

Abstract

We report the results of three empirical studies of fault detection and stability performance of the predicate-based BOR (Boolean Operator) testing strategy. BOR testing is used to develop test cases based on formal software specification, or based on the implementation code. We evaluated the BOR strategy with respect to some other strategies by using Boolean expressions and actual software. We applied it to software specification cause-effect graphs of a safety-related real-time control system, and to a set of N-version programs. We found that BOR testing is very effective at detecting faults in predicates, and that BOR-based approach has consistently better fault detection performance than branch testing, thorough (but informal) functional testing, simple state-based testing, and random testing. Our results indicate that BOR test selection strategy is practical and effective for detection of faulty predicates, and is suitable for generation of safety-sensitive test-cases.

1. Introduction

One common approach to software testing, referred to as predicate testing, is to require certain types of tests for each predicate (or condition) in a program or software specification. A number of predicate testing strategies have been proposed, including branch testing, domain testing, and others [How87, Bei90]. The field of evaluation of predicate testing strategies is very active and a number of theoretical [e.g., Cla85, Fra88, Kor88, Su89, Zei92, Fra93a, Wey94, Tai94] and experimental [e.g., Dur84, Tai87, Nta88, Su89, For93, Fra93b, Wey94, Tai94] studies are available. In this paper, we discuss several experimental evaluations of a set of predicate-based test generation and selection criteria called boolean operator (BOR) testing [Tai87, Tai93].

In the remainder of this section, we give some basic definitions. Section 2 provides an introduction to the predicate testing strategies related to our empirical studies. In section 3 we present an experiment based on Boolean expressions. In section 4 we discuss performance of BOR strategy in the context of a real-time safety-related application. In section 5 we report on the fault-detection properties of the BOR strategy when applied to a N-version program suite, and on the variability of the BOR coverage metric. Section 6 provides conclusions.

1.1 Definitions

A predicate can be simple or compound. A simple predicate is a boolean variable or a relational expression, possibly with one or more NOT ("~") operators. A relational expression is of the form

\[ E \langle rop \rangle E' \]

where \( E \) and \( E' \) are arithmetic expressions and \( \langle rop \rangle \) is one of six possible relational operators: "\(<\)", "\(\leq\)", "\(=\)", "\(\neq\)" (non-equality), "\(>\)", and "\(\geq\)". (Non-arithmetic expressions, such as character strings and sets, are not considered.) A compound predicate consists of at least one binary boolean operator, two or more operands, and possibly NOT operators and parentheses. The binary boolean operators allowed in a predicate include OR ("\(|\)"") and AND ("\(&\)"").

If a predicate is incorrect, then one or more of the following types of faults occur:

1. boolean operator fault (incorrect AND/OR operator or missing/extra NOT operator)
2. incorrect relational operator
3. incorrect parenthesis
4. incorrect boolean variable
5. incorrect arithmetic expression
6. extra binary operator (and operand)
7. missing binary operator (and operand)

In this paper, we focus on the detection of faults of types (1)-(3) and a special type of (4).

A test set for a predicate \( C \) is said to detect the existence of faults in \( C \) if an execution of \( C \) on at least one element of this test set produces (executes) an incorrect outcome (branch) of \( C \). A test set \( T \) for \( C \) is said to guarantee the detection of a certain type of faults in \( C \) if \( T \) detects the existence of such faults in \( C \),
provided that C does not contain faults of other types. Assume that predicate \( C^* \) has the same set of variables as C and is not equivalent to C. A test set T is said to distinguish C from \( C^* \) if C and \( C^* \) produce different outcomes on at least one element of T.

A test set T for a predicate C is said to satisfy a predicate testing strategy for C if the execution of C using T satisfies the requirements of this strategy. For two predicate testing strategies S and \( S' \), \( S \) is said to be stronger (weaker) than \( S' \) if any test set satisfying S for a predicate C also satisfies \( S' \) for C, but not vice versa. S and \( S' \) are said to be incomparable if neither of them is stronger than the other.

If any two test sets \( T_1 \) and \( T_2 \), chosen by S, are such that all elements of \( T_1 \) are successful when all elements of \( T_2 \) are successful, then S is reliable. In practice it may be that reliability of a strategy is not 1, i.e., some test sets chosen by strategy S do not perform, on the same program P, as well as other test sets chosen by the same strategy. If the criterion S can produce test sets that uncover all faults in program P, then S is valid for P. An ideal strategy would be reliable and valid for all P.

2. Predicate testing strategies

In this section we provide an overview of several predicate testing strategies that are related to our empirical studies. The predicate

\[
((E_1<E_2) \& (E_3\geq E_4)) \mid (E_5=E_6),
\]

where \( E_i \) (1\( \leq i \leq 6 \)), denotes an arithmetic expression. It is shown in Figure 1. We will call it \( C# \) and use it below to illustrate different predicate testing strategies.

![Figure 1. Binary decision diagram (BDD) of C#](image)

**Branch Testing**: This strategy requires that both the true and false branches of each predicate be executed (or covered) at least once.

**Complete Branch Testing**: For a compound predicate C, this strategy requires that the true and false branches of every simple or compound operand in C (including C itself) be executed at least once. Although complete branch testing is stronger than branch testing, it can be satisfied for a compound predicate by using two tests. The test set \{t1, t2\} shown below satisfies complete branch testing for \( C# \).

<table>
<thead>
<tr>
<th>( ((E_1&lt;E_2) &amp; (E_3\geq E_4)) \mid (E_5=E_6) )</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>t</td>
</tr>
<tr>
<td>t2</td>
<td>f</td>
</tr>
</tbody>
</table>

In the above table, the values of t1 and t2 are not given. Instead, each of t1 and t2 is specified in terms of the outcome ("t" for true and "f" for false) of each simple predicate in \( C# \). t1 is a test to make \((E_1<E_2)\) true, \((E_3\geq E_4)\) true, and \((E_5=E_6)\) true. Similarly, t2 is a test to make \((E_1<E_2)\) false, \((E_3\geq E_4)\) false, and \((E_5=E_6)\) false. t1 and t2 are said to satisfy constraints \((t,t,t)\) and \((f,f,f)\), respectively, for \( C# \). The outcome of \( C# \) on t1 (t2) can be determined from \( C# \) and the constraint satisfied by t1 (t2). The constraint set \{\((t,t,t), (f,f,f)\)\} cannot distinguish \( C# \) from the following predicates:

\[
( (E_1<E_2) \mid (E_3\geq E_4)) \mid (E_5=E_6),
\]

which differ from \( C# \) in boolean operators only.

**Exhaustive Branch Testing**: For a compound predicate C, this strategy requires that all combinations of true and false branches of every simple predicate in C be executed at least once. If C consists of \( n, n>0 \), AND/OR operators, then exhaustive branch testing for C requires \( 2^{(n+1)} \) constraints and is impractical when \( n \) is large.

**Boolean Operator Testing** (or BOR Testing): A test (constraint) set \( T \) for a compound predicate C is said to be a BOR test (constraint) set if \( T \) guarantees the detection of boolean operator faults in C, including incorrect AND/OR operators and missing or extra NOT operators. The test set \{t3, t4, t5, t6\} shown below is a BOR test set for \( C# \):

<table>
<thead>
<tr>
<th>( ((E_1&lt;E_2) &amp; (E_3\geq E_4)) \mid (E_5=E_6) )</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>t3</td>
<td>t</td>
</tr>
<tr>
<td>t4</td>
<td>t</td>
</tr>
<tr>
<td>t5</td>
<td>f</td>
</tr>
<tr>
<td>t6</td>
<td>f</td>
</tr>
</tbody>
</table>

The constraint set \{\((t,t,f), (t,f,t), (t,f,f), (f,t,f)\)\} is called a BOR constraint set for \( C# \). An algorithm, called BOR_GEN, for generating a minimum BOR constraint set for a compound predicate was given in [Tai93]. For a predicate with \( n, n>0 \), AND/OR operators, its minimum BOR constraint set contains at most \( n+2 \) constraints. For
a compound predicate, BOR testing requires the coverage of a minimum BOR constraint set for this predicate.

**BDD Path Testing:** For a compound predicate C, this strategy requires that all paths in the BDD (binary decision diagram) of C be covered at least once. Figure 1 shows the BDD of C#. The notion of BDDs has been used in the representation, verification and testing of boolean functions and logical circuits [Ake78, Bry86]. The BDD of a compound predicate is equivalent to the compound predicate modified by replacing "AND" and "OR" with "AND THEN" and "OR ELSE", respectively. (For C1 AND THEN C2, if C1 is false, then C2 is ignored. For C1 OR ELSE C2, if C1 is true, then C2 is ignored.) The test set \{t7, t8, t9, t10\} shown below satisfies BDD path testing for C#.

**Table 3. BDD Path Testing**

<table>
<thead>
<tr>
<th></th>
<th>(E1&lt;E2) &amp; (E3≥E4)</th>
<th>(E5=E6)</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>t7</td>
<td>t</td>
<td>t</td>
<td>*</td>
</tr>
<tr>
<td>t8</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>t9</td>
<td>t</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t10</td>
<td>f</td>
<td>*</td>
<td>f</td>
</tr>
</tbody>
</table>

In the above table, "*" denotes "don't care". For example, the constraint (t,t,*), satisfied by t3 is to make both (E1<E2) and (E3≥E4) true, with no constraint on E5 and E6. Since "*" denotes "don't care," the value of ("t"&"*") is uncertain and so is that of ("f"&"*"). The constraint set \{(t,t,*), (t,f,t), (t,f,f), (f,*),f\} does not guarantee to distinguish C# from the following two predicate

\[
((E1<E2) \| (E3>E4)) \& (E5=E6)
\]

which differ from C# in boolean operators only. For a predicate with n, n>0, AND/OR operators, BDD path testing strategy requires (n+2) or more, up to O(2^n), constraints.

**Meaningful Impact Strategies:** In [Wey94] a family of strategies for generating tests for a boolean expression was described. Below we show a test set for C# according to one of these strategies, called the MIN strategy.

**Table 4. MIN Testing**

<table>
<thead>
<tr>
<th></th>
<th>(E1&lt;E2) &amp; (E3≥E4)</th>
<th>(E5=E6)</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>t27</td>
<td>t</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>t28</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>t29</td>
<td>f</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>t30</td>
<td>t</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

To explain the generation of the above test set, consider C# as (a&b)+c, where a, b, and c denote (E1<E2), (E3≥E4), and (E5=E6), respectively. Both t27 and t28 make C# true. However, t27 makes (a&b) true and c false, while t28 makes (a&b) false and c true. Both t29 and t30 make C# false. However, t29 makes (¬a&b) true and t30 (a&¬b) true and both make ¬c true. The Meaningful Impact strategies are based on a sum-of-product form of a boolean expression. Empirical studies showed that these strategies are very effective. The authors report that the number of tests produced for a boolean expression varies from about 6% to about 48% of the number of exhaustive tests, depending on the strategy used, but they do not provide an upper bound for the number of test cases.

**Relational Operator Testing:** For a relational expression, say (E rop E"), this strategy requires three tests satisfying the following requirements [How82]:

1. one test makes E > E",
2. one test makes E < E",
3. one test makes E = E".

If rop is incorrect and E and E" are correct, then this strategy guarantees the detection of the incorrect rop.

Relation operator testing is stronger than branch testing. For a compound predicate C, relational operator testing for each relational expression in C does not guarantee the detection of boolean operator faults. The test set \{t18, t19, t20\} shown below for C# satisfies relational operator testing for each relational expression in C#.

**Table 5. Relational Operator Testing**

<table>
<thead>
<tr>
<th></th>
<th>(E1&lt;E2) &amp; (E3≥E4)</th>
<th>(E5=E6)</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>t18</td>
<td>=</td>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>t19</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>t20</td>
<td>&gt;</td>
<td>=</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

In the above table, a constraint for a relational expression is "<", "=" or ">", indicating that the left-hand side of the expression is less than, equal to, or greater than, respectively, the right-hand side of the expression. For example, the constraint (=,>,=) satisfied by t18 is to makes E1=E2, E3>E4, and E5=E6. The constraint set \{(=,>,=), (<,<,<), (<,<,<), (<,<,<), (=,=,>), (>,=,>)} satisfies relational operator testing for each relational expression in C#, but it cannot distinguish C# from the following predicates

\[
((E1<E2) \& (E3>E4)) \& (E5=E6),
\]

\[
((E1=E2) \& (E3=E4)) \& (E5=E6),
\]

\[
((E1<E2) \& (E3=E4)) \& (E5=E6),
\]

\[
((E1=E2) \& (E3=E4)) \& (E5=E6),
\]

which differ from C# in relational operators only.

**Boolean and Relational Operator Testing (or BRO Testing):** A test (constraint) set T for a compound predicate C is said to be a BRO test (constraint) set for C if T guarantees the detection of boolean and/or relational operator faults in C, including incorrect AND/OR operators, missing or extra NOT operators, and incorrect relational operators. The test set \{t21, ..., t26\} shown in Table 6 below satisfies BRO testing for C# and is called a BRO test set for C#.

**Table 6. BRO Testing**

<table>
<thead>
<tr>
<th></th>
<th>(E1&lt;E2) &amp; (E3≥E4)</th>
<th>(E5=E6)</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>t21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t26</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The constraint set \{(<,<,<), (<,<,<), (<,<,<), (<,<,<), (=,=,>), (>,=,>)} is called a **BRO constraint set** for C#. A algorithm, called BRO_GEN for generating a minimum BRO constraint set for a compound predicate was given in [Tai93]. For a predicate with n, n>0,
AND/OR operators, its minimum BRO constraint set contains at most $2^n+3$ constraints. For a compound predicate, BRO testing requires the coverage of a minimum BRO constraint set for this predicate.

<table>
<thead>
<tr>
<th>Table 6. BRO Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((E_1 &lt; E_2) &amp; &amp; (E_3 &gt; E_4))</td>
</tr>
<tr>
<td>t21</td>
</tr>
<tr>
<td>t22</td>
</tr>
<tr>
<td>t23</td>
</tr>
<tr>
<td>t24</td>
</tr>
<tr>
<td>t25</td>
</tr>
<tr>
<td>t26</td>
</tr>
</tbody>
</table>

It is important to point out the possible existence of infeasible constraints for a predicate. Consider the predicate C#. If a constraint for C# can never be covered by any test for C#, it is said to be an infeasible constraint for C#. More discussion on infeasible constraints can be found in [Tai93].

3. A comparison between BOR, BDD Path, and Branch testing by using Boolean expressions

For a predicate with n, n\(\geq 0\), AND/OR operators, branch testing requires 2 tests, BOR testing n+2 or less tests, and BDD Path testing n+2 or more, up to O(2^n), tests. BOR and BDD path testing is stronger than branch testing, and are incomparable with each other.

Let a singular boolean expression (SBE) be a boolean expression in which each boolean variable occurs only once. The reason for using SBEs is to focus on the detection of boolean operator faults and incorrect parentheses. (Note that a BOR constraint set for a predicate guarantees the detection of boolean operator faults only if no other types of faults exist.) A constraint set for a boolean expression is called a test set since each constraint is actually a test.

An empirical study was conducted to compare the effectiveness of these three predicate testing strategies. The following sets of SBEs were constructed:

(a) a set of 48 mutually non-equivalent SBEs with three variables,
(b) a set of 366 mutually non-equivalent SBEs with four variables, and
(c) a set of 2624 mutually non-equivalent SBEs with five variables.

In each of (a), (b), and (c), these SBEs differ from each other in boolean operators and/or parentheses. The following table shows, for each of (a), (b), and (c), the average fault detection ratio (expressed as percentage of detected faults) for BOR, BDD Path, and Branch testing, respectively. (For branch testing of a boolean expression, two tests were chosen to satisfy complete branch testing.)

Table 7. Fault detection ratios

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOR testing</td>
<td>99.3%</td>
<td>99.7%</td>
<td>99.9%</td>
</tr>
<tr>
<td>BDD Path testing</td>
<td>90.1%</td>
<td>90.4%</td>
<td>91.9%</td>
</tr>
<tr>
<td>Branch testing</td>
<td>72.2%</td>
<td>72.5%</td>
<td>72.9%</td>
</tr>
</tbody>
</table>

Our results show that

(1) branch testing is far less effective than BOR testing for fault detection.
(2) BOR testing performs better than BDD path testing. Although BDD path testing requires at least as many tests as BOR testing, its use of "don't care" reduces its effectiveness for fault detection.
(3) similar to the experimental comparisons with the Elmendorf's algorithm [Elm78] reported in [Tai94], BOR testing almost guarantees the detection of boolean operator faults and incorrect parentheses.

4. Applying BOR testing to a safety-related real-time system

In this section, we describe an application of BOR testing to control software for a simplified real-time boiler control and monitoring system. The specifications were developed as part of the generic problem exercise conducted for the 1993 International Workshop on the Design and Review of Software Controlled Safety-Related Systems [IRR93]. A version of the boiler control and monitoring system was developed at North Carolina State University [Vou93a]. This software was used in this empirical study. A brief description of the boiler system is given below.

The simplified boiler system used in the study consists of a natural-gas fired water-tube boiler producing saturated steam. The steam flow may vary rapidly and irregularly between zero and maximum, following a varying external demand. The water level in the boiler is regulated by the inflow of feedwater. The water level must be kept between an upper and lower limits. If the water level is above the upper limit, water will be carried over into the steam flow and cause damage. If the water level is below the lower limit, boiler tubes will dry out and may overheat and burst. If the control of water level is lost, the boiler is shut down. The water level and the steam flow are measured by an instrumentation system, which reports sensor values. The readings from sensors are transmitted over an intrinsically unreliable communication link to the control program. This control program is expected to perform the following tasks:

(a) To regulate the water level by controlling the inflow of feedwater by appropriately turning pumps on, or off, at required instances.
(b) To diagnose and isolate all the potential errors and issue a correction/repair request if any are discovered.
(c) To display at all times "best estimates" of various
readings for the boiler operator.

(d) To accept any appropriate operator commands.

During the development of the boiler system at NCSU, the original, informal, specification of the system [IRR93] was re-written in terms of a number of extended finite-state machines\(^1\) (EFSMs), and test suites for the unit, integration and system testing of the boiler system were constructed according to the boiler's EFSM specification [Par93]. The EFSM specification-based test suites were then developed to ensure thorough testing of the boiler system. In addition to the coverage of every state and branch of individual EFSMs, great effort was made to construct additional test cases to cover special event situations. However, no well defined strategies were used for testing combinations of EFSM predicates, that is combinations of states.

We considered both specification- and program-based BOR testing of the boiler system. Since the complete boiler system has about 4,500 lines of C code, and there was no automatic tool available for measuring the BOR coverage of the C code\(^2\), we chose to experiment only with the most critical boiler system effect, the "boiler shutdown" effect.

For this effect we derived a specification cause-effect graph (CEG) and analyzed its implementation (code) CEG. The notion of CEGs was developed for system specification and test generation [Mye79]. A CEG is a graphical notation for describing logical relationships among causes and effects. A cause is an input condition, an effect is an output condition, and logical operators include AND, OR, NOT, and others.

### 4.1 A cause-effect graph for the shutdown effect

The specification-CEG for the shutdown effect, referred to as the shutdown CEG, is organized in five levels. The level 1 (the highest level) specification-CEG for boiler shutdown is shown in Figure 2. The annotations for nodes in level 1 CEG are given below.

- E - boiler shutdown
- C221 - externally initiated
- C220 - internally initiated
- C202 - operator initiated
- C203 - instrumentation system initiated
- C201 - bad startup
- C200 - operational failure
- C197 - confirmed keystroke entry
- C198 - confirmed "shutnow" message
- C196 - multiple pumps failure (more than one)
- C195 - water level meter failure during startup
- C194 - steam rate meter failure during startup
- C193 - communication link failure
- C192 - instrumentation system failure
- C191 - C180 and C181
- C190 - water level out of range
- C180 - water level meter failure during operation
- C181 - steam rate meter failure during operation

![Figure 2. Level 1 Specification Cause-Effect Graph for Boiler Control and Monitoring System](image)

The cause nodes of the level 1 specification-CEG, including C180, C181, C190, and C192 through C198, are effect nodes of level 2 CEGs. Similarly, some of the cause nodes of level 2 CEGs are effect nodes of level 3 CEGs, and so on. CEGs of level 2 through 5 are not shown in this paper.

### 4.2 BOR testing of the specification-CEG

From the EFSM test cases developed previously, we selected those related to the shutdown effect. The selected test set, referred to as the shutdown test set, contained 372 test cases. We then measured the BOR coverage that this set offered for the specification-CEG and for the implementation-CEG of one module, or C program.

We used the BOR_COV algorithm to measure the BOR coverage of the shutdown specification-CEG by the shutdown test set [Tai94].

Of the 372 tests in the shutdown test set, 59 tests...
(about 1/6 of the total) were found to be redundant. Also, 24 more constraints were needed for BOR testing. So the BOR coverage of the shutdown CEG by the shutdown test set was \((372-59)/(372-59+24) = 0.928\). Most of the redundant tests dealt with pump/flow monitor combinations. However, most of the additional tests needed for BOR testing also dealt with pump/flow monitor combinations.

Furthermore, the EFSM shutdown test set was constructed by three persons with a total of approximately 100 person-hours. The shutdown specification-CEG was constructed by one person in about 20 hours, and the CEG-based test-case generation can be automated. Also, CEGs can be analyzed for the detection of ambiguities and inconsistencies in system specification [Mye79, Bei90].

It is also worth noting that the additional tests needed for 100% BOR coverage of the shutdown CEG detected a bug in the boiler's implementation (see next section).

It is obvious that use of CEGs for software specification and BOR strategy for test generation has significant advantages.

### 4.3 BOR coverage of a module in the boiler's implementation

We chose one module in the boiler control software implementation for the measurement of BOR coverage. The selected module deals with the shutdown effect.\(^3\) and contains 360 C language statements and 34 predicates, of which 21 are simple predicates (that is, there are no AND/OR operators), and the remainder are compound predicates with one AND/OR operator. We manually transformed this module for the measurement of BOR coverage, and generated 81 BOR constraints for the predicates of this module. (Two constraints are needed for each simple predicate, and three constraints for each compound predicate with one AND/OR operator.)

The shutdown test set used was to execute the selected module\(^4\). Two of the 81 constraints were not covered by the shutdown test set. So the BOR coverage of the selected module by the shutdown test set is \(79/81 = 0.975\). When the two uncovered constraints were further investigated, a bug was discovered in the selected module. This bug would have been uncovered if the selected module had been tested using the 100% BOR coverage criterion.

### 4.4 Safety

It is interesting to note that cause-effect graphs are not that different from FAULT-TREES when the effect is chosen to be the undesirable safety-critical failure. Then, the causes become basic triggering events for such failures. It is also interesting to note that every intermediate node in a cause-effect graph, as well as a fault-tree, is in essence a predicate, i.e., an expression that determines the course of action depending on whether it is true of false. Hence, predicate-based analysis and verification can be used to ascertain correctness of the fault-trees as well as develop safety-sensitive test-suites for implementation testing.

For example, in the "boiler" system CORRECT boiler shutdown is the single most safety critical effect in the system, and it was chosen for detailed CEG-BOR analysis. The critical failure is the one where the shutdown (function) does not occur when it should. There are two approaches to analysis:

1. Construct a fault-tree where NO_SHUTDOWN is the failure. This failure occurs when shutdown fails to take place for critical conditions.
2. Construct a cause-effect graph where the protective EFFECT of SHUTDOWN occurs, given that critical causes have occurred. This approach requires construction of a safety test-bed.

We chose the second approach, since that allowed us to test for things that SHOULD have happened given appropriate (bad) input conditions as opposed to having to prove that a failure NEVER occurs. In this way our testing, and our test cases, test for safety and also evaluate the reliability of that safety feature by identifying how often it works and in what situations it fails.

However, the technique does require that safety-critical effects are explicitly built into the specifications as predicates and explicit PROTECTIVE effects (we then look for execution of the protective exception as opposed to lack of occurrence of a failure). In essence, we transform a fault-tree into a "complementary" image of protective functions that guard against the critical failure and ensure execution of these protective functions under all adverse conditions. As a side-effect we get to evaluate the "false-alarm" rate, i.e., the frequency of execution of protective features when there were no actual failures. Excessive "false-alarm" rate can discourage use of the safety devices (e.g., problems with the collision avoidance systems currently installed on commercial airliners).

We see that, given properly constructed CEG, the test cases developed using the BOR strategy can be safety-oriented, and can have better fault-detection properties than the "traditional" testing approaches.

### 5. Testing of a N-version program set

Test suites can be evaluated for compliance with some test-stopping criterion, such as full branch coverage or BOR coverage. However, it is less clear that full compliance with a testing criterion at the specification stage provides any guarantees at the software implementation stage. The reason is that a specification CEG may be implemented in many functionally

---

\(^3\)Note that the selected module is not the only module dealing with the shutdown effect.

\(^4\)Remember that only a portion of the shutdown test set is meant for the selected module.
equivalent ways. Therefore, it is important to evaluate a specification-based testing strategy for its effectiveness over the implementation population.

In this section, we report an empirical study in which we applied specification-based BOR testing to a set of N-version programs\(^5\) written to the same specification in Pascal. We wished to evaluate the behavior of the BOR strategy over a sample of functionally equivalent versions and thus provide an insight into the self-consistency and validity of the strategy. A tool called BGG, developed at North Carolina State University to measure the test coverage of statements, branches, and various types of data flow metrics for Pascal programs [Vou89], was extended to generate BOR and BRO constraint sets for predicates in a Pascal program, and to measure the coverage of BOR and BRO constraints in a Pascal program according to a given test set.

Six functionally equivalent programs were produced as part of another study [Vou86a, Vou86b]. Programs solve a navigational problem, an extended version of the "Earth Satellite Problem" used by Nagle at al. [Nag82]. They were written independently by graduate level students to the same specification. The programming language was Pascal. The size of the Pascal programs ranged from 400 to 800 Pascal statements. Acceptance testing of the developed software involved both random and special functional test cases. "White-box", or structural, testing was not part of the acceptance testing. All acceptance test data were generated on the basis of functional program specifications. Special consideration was given to extremal and special values, explicit or implicit, in the specification (boundaries, singularities, etc.), as well as in the known problem solution algorithms. Random data were generated using a uniform distribution for all input parameters. Failures observed in the components are described in [Vou86b]. For technical reasons, only five of those program were used in the current experiment.

5.1 An evaluation of BOR testing

We built the "Earth Satellite Problem" CEG using the requirements specifications and then selected test cases according to the BOR criterion. During the construction of the of the CEG we detected a specification ambiguity and built its resolution into the CEG [Par94]. This helped detect resulting problems that have propagated into the implementations.

Satisfaction of the BOR criterion required generation of 43 test cases. Full CEG coverage would have required 65 test cases. Some of the test cases dealing with precision in the software could not be derived from CEG of the specification alone, but the requirements were implicitly mentioned and thus they were deliberately worked into the CEG. (This is the subjective/educated development part of the CEG and accounts for possible diversity in specification-CEGs).

The coverage provided by the 43 BOR test cases was practically identical to that provided by the functional test suite of 103 test cases. Table 8 summarizes the average coverage measures over these five programs, and is further discussed in the next sub-section. Since CEG-based BOR (CEG-BOR) testing required about half as many test cases as the "traditional" functional test-suite, the specification-based use of BOR test selection may have considerable advantages in terms of cost.

![Figure 3 Fault detection dynamics for program L2.](image)

Of course, unless the fault-detection power of the BOR approach at least matches that of the more informal strategies, the BOR strategy would not be very useful in practice. In our experiments the BOR data set detected all known faults in all five programs. In contrast, in the case of program L2 and L3 neither functional testing alone nor random testing alone detected all faults found in these programs.

The experiment specific fault-detection dynamics is illustrated in Figure 3 for program L2 data. It shows the plot of the total number of faults detected by the BOR, functional and random testing versus the number of test cases run (in the order they were generated and used in the experiments). We see that BOR detects all faults that functional and random testing did together, and one more, and did it that with a smaller number of test cases than either. Similar results were obtained for other programs.

\(^5\)N-version programming is a technique for increasing software reliability, is to develop multiple versions of a software system for a given specification and then execute these versions at the same time to compare their results.
5.2 Coverage performance of Specification-based Testing

This part of the experiment was used to investigate the impact of implementation diversity on three specification-based testing strategies. Each of the five programs (using individual drivers) was tested by two three test sets: one contained 1,000 randomly selected tests, the second contained 103 manually designed functional tests (i.e., special value functional tests), and the third was composed of 43 test cases developed using the CEG-BOR strategy. For each program we measured the coverages of p-uses [Fra88, Wey88], BRO constraints, branches and statements.

The results are illustrated in Figures 4 through 9, and in Table 8. The programs are marked L1 through L5. In all figures, we see the usual metric saturation behavior characteristic of single strategy testing approaches, and one other interesting phenomenon — the noise the implementation diversity causes in the code coverage.

The design and implementation transformations of the single requirements specifications that all versions used, resulted in diverse but functionally equivalent programs, and, as would be expected, these programs have different implementation CEGs. From the graphs, we see that this reflects as considerable variability in the coverage achieved by a particular test-set over this population of functionally equivalent programs — for example, random testing...
results in BRO coverage that differs by as much as 20% between L2 and L4 versions. Similar variance in coverage metrics has been seen in experiments with other software [Vou93b]. Interestingly, the "best behaved" metric appears to be p-use. Presently, it is not clear whether this is because of the data-flow nature of this metric, or because the usually large number of p-use constructs acts as a normalizing factor across the program population.

This indicates that any type of reliability modeling that is based on coverage metrics will have to account for this variability. For instance, an implication is that a test evaluation of strategy based solely on achievement of partial coverage (e.g., 90% of branches) may have no practical meaning in terms of fault detection guarantees and reliability, unless it is transformed into a canonical form that is valid across the population.

Using BRO-metric as an example, we see that

1. The coverages of BRO constraints and branches for a program increase very rapidly by the first ten random tests or the first twenty functional tests. Afterwards, these coverages increase very slowly. In fact, the last 900 tests of the 1000 random tests provide insignificant increase of coverage.

2. Functional testing provides 20% or more coverage of BRO constraints or branches than random testing. This result indicates that although random testing is easy to do, it must be supplemented with functional testing.

3. For a program, the same test set provides coverage of BRO constraints that is about 15-20% less than that of branches. This result provides some indication of relative "strength" between BRO and branch testing.

6. Conclusions

The objective of this work was to experimentally evaluate the BOR strategy for specification- and program-based testing. We were especially interested in three properties. The ability of BOR testing to detect faults within software, its stability (that is, the ability to consistently and reliably provide this fault detection capability across different software application areas), and its possible application to safety-sensitive software.

Our experiments indicate that the BOR approach is probably very effective in detecting faults at fraction of the cost. It is considerably better than branch testing, and comparable with more elaborate and expensive predicate based methodologies, such as Elmendorf's algorithm. Preliminary experiments with actual software indicate that BOR based testing, when developed using software specifications, has fault detection properties which are at least comparable to those of thorough, but informal, functional and random testing, but requires far fewer test cases. This behavior was observed consistently in two application and over five functionally equivalent programs.

However, none of the specification-based testing strategies we investigated provided full coverage of the implementation codes, even at the statement level, although the coverage achieved by the CEG-BOR provided was always as good or better than that provided by the functional tests. In addition, CEG based analysis and BOR testing can be adapted to evaluate safety fault-trees and there are indications that CEG-BOR may represent a good way of generating safety-sensitive test cases.

The work on the application of BOR/BRO strategies to safety-critical applications continues. We are also in the process of applying the CEG BOR/BRO methodology to
a large real-life data-base product to validate the scalability of the approach.

7. References


