Evaluation of a predicate-based software testing strategy

by K.-C. Tai
M. A. Vouk
A. M. Paradkar
P. Lu

In this paper, we report the results of four empirical studies for evaluating a predicate-based software testing strategy, called BOR (Boolean operator) testing. The BOR testing strategy focuses on the detection of Boolean operator faults in a predicate, including incorrect AND/OR operators and missing or extra NOT operators. Our empirical studies involved comparisons of BOR testing with several other predicate-based testing strategies, using Boolean expressions, a real-time control system, and a set of N-version programs. For program-based test generation, BOR testing was applied to predicates in a program. For specification-based test generation, BOR testing was applied to cause-effect graphs representing software specification. The results of our studies indicate that BOR testing is practical and effective for both specification- and program-based test generation.

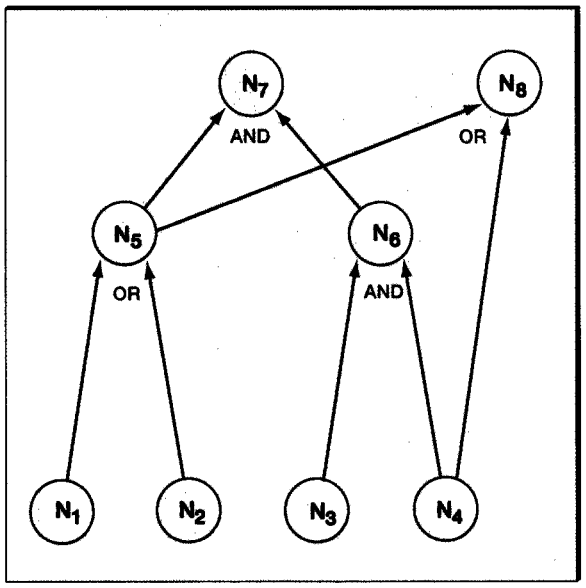
The testing activities of a software project usually take about half of the total cost.[1,2] A major problem in testing a program is how to reduce the effort of generating a test set that is effective for detecting faults in the program. One approach to software testing, referred to as *predicate testing*, is to require certain types of tests for each predicate (or condition) in a program or software specification. One commonly used predicate testing strategy is branch testing, which requires that the true and false branches of a predicate be executed at least once.

As the complexity of software increases, so does the number of compound predicates, which are predicates with one or more AND/OR operators, in software specification and implementation. Recently a predicate testing strategy, called BOR (from *B*oolean *o*perator) testing, was proposed for testing compound predicates.[3] BOR testing focuses on the detection of Boolean operator faults in a predicate, including incorrect AND/OR operators and missing or extra NOT operators; it can be applied to either specification- or program-based test generation. In this paper, we report the results of four empirical studies for evaluating the BOR testing strategy.[4]

The remainder of this section contains basic definitions. The next section discusses several predicate testing strategies and illustrates the difficulty of testing compound predicates. The section thereafter introduces the BOR testing strategy. The remaining sections of the paper show the results of four empirical studies, two of them based on the use of Boolean expressions, one based on a real-time boiler control and monitoring system, and one based on a set of *N*-version programs.

A predicate is either a simple or compound predicate. A *simple predicate* is a Boolean variable or

Figure 1 A cause-effect graph



a relational expression, possibly with one or more NOT ("~") operators. A *relational expression* is of the form

E_1 <rop> E_2

where E_1 and E_2 are arithmetic expressions and <rop> is one of six possible relational operators: "<", "<=", "=", "≠", ">", and ">=". (Non-arithmetic expressions, such as character strings and sets, are not considered in this paper.) 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 considered in this paper include OR ("|") and AND ("&"). A *Boolean expression* is a predicate without relational expressions. In this paper, B_i, $i>0$, denotes a Boolean variable, and E_i denotes an arithmetic expression.

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

1. Boolean operator fault (incorrect AND/OR operator or missing or extra NOT operator)
2. Incorrect relational operator
3. Incorrect parentheses
4. Incorrect arithmetic expression
5. Incorrect Boolean variable
6. Extra binary operator and its operands
7. Missing binary operator and its operands

An incorrect predicate contains either a single fault or multiple faults of the same or different types. 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 an incorrect outcome of C. A test set T for C is said to *guarantee the detection* of certain types of faults in C, if T can detect 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. As an example, the test set {(t,t), (t,f), (f,t)}, where "t" and "f" denote "true" and "false," respectively, distinguishes $(B_1\&B_2)$ from other Boolean expressions that differ from $(B_1\&B_2)$ in Boolean operators only. The set {(t,t), (t,f), (f,t)} is said to guarantee the detection of Boolean operation faults in $(B_1\&B_2)$.

A test set T for a predicate C is said to *satisfy* a predicate testing criterion (or strategy)[5] for C, if the executions of C using T satisfy the requirements of this criterion. A predicate testing criterion (or strategy) is said to *guarantee the detection* of certain types of faults in predicate C, if any test set satisfying this criterion for C can detect the existence of such faults in C. For two predicate testing criteria (or strategies) S and S′, S is said to be stronger than S′, if any test set satisfying S for a predicate also satisfies S′ for the same predicate, but not vice versa.

A cause-effect graph (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. The notion of CEGs was developed for system specification and test generation.[1,6] A test set for a cause-effect graph can be used to verify this graph as well as any program that implements this graph. Figure 1 shows a CEG, with nodes N_1 through N_4 denoting causes, nodes N_5 and N_6 intermediate nodes, and nodes N_7 and N_8 effects. The CEG in Figure 1 can be viewed as a collection of two predicates: $((N_1|N_2)\&(N_3\&N_4))$ for N_7 and $((N_1|N_2)|N_4))$ for N_8.

Predicate testing strategies

This section discusses several predicate testing strategies and illustrates the difficulty of testing compound predicates. More details of predicate testing strategies can be found elsewhere.[1,7] The following predicate

$$((E_1 < E_2) \,\&\, (E_3 >= E_4)) \mid (E_5 = E_6)$$

where E_1 through E_6 denote arithmetic expressions, is denoted as C# and used later for illustration.

Branch testing. This strategy requires that the true and false branches of a predicate be executed (or covered) at least once. The number of tests required for a predicate is two and does not depend upon the complexity of this predicate.

Complete branch testing. This strategy requires that, for a compound predicate C, the true and false branches of every simple or compound predicate in C (including C itself) be executed at least once. Although complete branch testing is stronger than branch testing, the former usually can be satisfied for any compound predicate by using two tests.[8] Thus, complete branch testing is not necessarily more effective than branch testing for fault detection.

The test set $\{t_1, t_2\}$ shown in Table 1 satisfies complete branch testing for C#. In the table, the values of t_1 and t_2 are not given. Instead, each of t_1 and t_2 is specified in terms of the outcome ("t" or "f") of each relational expression in C#. Test t_1 makes $(E_1<E_2)$ true, $(E_3>=E_4)$ true, and $(E_5=E_6)$ true. Similarly, test t_2 makes $(E_1<E_2)$ false, $(E_3>=E_4)$ false, and $(E_5=E_6)$ false. t_1 and t_2 are said to satisfy or cover constraints (t,t,t) and (f,f,f), respectively, for C#. The constraint set {(t,t,t), (f,f,f)} is said to satisfy complete branch testing for C#. Note that $\{t_1,t_2\}$ does not distinguish C# from the following predicates, which differ from C# in Boolean operators only:

$$((E_1<E_2) \mid (E_3>=E_4)) \mid (E_5=E_6)$$

$$((E_1<E_2) \mid (E_3>=E_4)) \,\&\, (E_5=E_6)$$

$$(\sim(E_1<E_2) \mid (E_3>=E_4)) \,\&\, (E_5=E_6)$$

$$((E_1<E_2) \mid \sim(E_3>=E_4)) \,\&\, (E_5=E_6)$$

$$((E_1<E_2) \,\&\, (E_3>=E_4)) \,\&\, (E_5=E_6)$$

Table 1 Test set $\{t_1, t_2\}$

	$((E_1<E_2)$	&	$(E_3>=E_4))$	|	$(E_5=E_6)$	outcome of C#
t_1	t		t		t	t
t_2	f		f		f	f

Table 2 Test set $\{t_3, t_4, t_5\}$

	$((E_1<E_2)$	&	$(E_3>=E_4))$	|	$(E_5=E_6)$	outcome of C#
t_3	=		>		=	t
t_4	<		<		<	f
t_5	>		=		>	f

(Each of the above predicates produces the same results on t_1 and t_2 as C#.)

Relational operator testing. For a relational expression, say (E <rop> E′), this testing strategy requires three tests satisfying the following requirements:[9,10] (1) one test makes E > E′, (2) one test makes E < E′, and (3) one test makes E = E′. If <rop> is incorrect and E and E′ are correct, this strategy guarantees the detection of the incorrect <rop>. (For a relational expression, relational operator testing is stronger than branch testing.) For a compound predicate C containing multiple relational expressions, one intuitive approach is to require relational operator testing for each relational expression in C. However, this requirement does not guarantee the detection of incorrect relational operators in C.

The test set $\{t_3, t_4, t_5\}$ shown in Table 2 for C# satisfies relational operator testing for each relational expression in C#. In Table 2, each of t_3, t_4, and t_5 is specified in terms of "<", "=", or ">" for a relational expression, indicating that the left side of the expression is less than, equal to, or greater than, respectively, the right side of the expression. For example, t_3 makes $E_1=E_2$, $E_3>E_4$, and $E_5=E_6$, and is denoted by the constraint (=,>,=). The constraint set {(=,>,=), (<,<,<), (>,=,>)} is said to satisfy relational operator testing for relational expressions in C#. $\{t_3, t_4, t_5\}$ does not distinguish C# from the following predicates, which differ from C# in relational operators only:

$$((E_1<=E_2) \,\&\, (E_3>=E_4)) \mid (E_5=E_6)$$

$$((E_1= E_2) \,\&\, (E_3>=E_4)) \mid (E_5=E_6)$$

$$((E_1< E_2) \,\&\, (E_3> E_4)) \mid (E_5=E_6)$$

$$((E_1= E_2) \,\&\, (E_3> E_4)) \mid (E_5=E_6)$$

Figure 2 Syntax tree for $((E_1<E_2)\ \&\ (E_3>=E_4))\ |\ (E_5=E_6)$

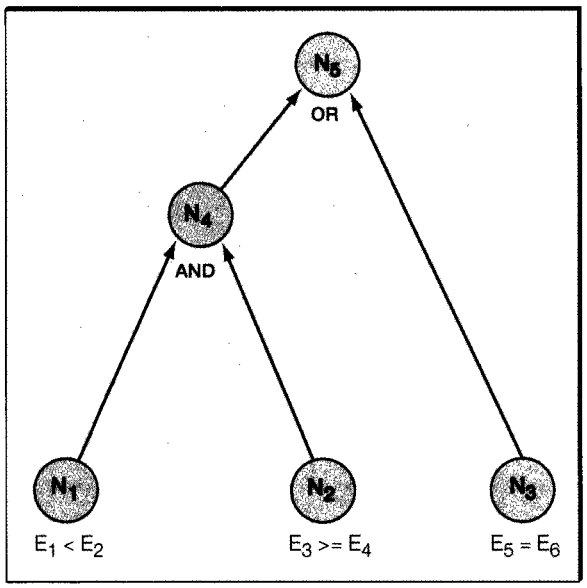


Table 3 Constraint set $\{t_6, \ldots, t_{12}\}$

	$((E_1<E_2)$	&	$(E_3>=E_4))$	|	$(E_5=E_6)$	outcome of C#
t_6	t		t		f	t
t_7	t		f		t	t
t_8	f		t		t	t
t_9	f		f		t	t
t_{10}	t		f		f	f
t_{11}	f		t		f	f
t_{12}	f		f		f	f

Exhaustive testing. The examples shown above illustrate the following two problems in testing compound predicates: (i) detection of (single or multiple) Boolean operator faults, (ii) detection of (single or multiple) relational operator faults. For a compound predicate C, if we require that all combinations of "t" and "f" for each simple predicate in C be executed at least once, then problem i is solved. If we require that all combinations of "<", "=", and ">" for each relational expression in C be executed at least once, then problem ii is solved. Assume that C consists of $n>0$ AND/OR operators. The exhaustive testing solution to problem i requires $2**(n+1)$ tests and to problem ii $3**(n+1)$ tests. Thus, exhaustive testing is not practical.

Elmendorf's strategy. Elmendorf developed a test generation algorithm for cause-effect graphs.[1,6] Since a cause-effect graph is a collection of compound predicates, Elmendorf's strategy can be applied to generate tests for a compound predicate. Below we show the application of this strategy to generate tests for C#. Figure 2 shows the syntax tree[11] for C#, which contains

- Three leaf nodes N_1, N_2, and N_3, which correspond to $(E_1<E_2)$, $(E_3>=E_4)$, and $(E_5=E_6)$, respectively
- Node N_4, which corresponds to the "&" operator
- Node N_5, which corresponds to the "|" operator

The nodes in the syntax tree for C# are visited from the root node to leaf nodes. For node N_5, inputs (t,f), (f,t), and (f,f) are selected, with the first element of each input being the output of N_4 and the second element of each input being the output of $E_5=E_6$. For node N_4 with output value "t", input (t,t) is selected, and for node N_4 with output value "f", inputs (t,f), (f,t), and (f,f) are selected, with the first element of each input being the output of $(E_1<E_2)$ and the second element of each input being the output of $(E_3>=E_4)$. Thus, Elmendorf's strategy generates the constraint set $\{t_6, \ldots, t_{12}\}$ for C# shown in Table 3.

Equivalence partitioning testing. Yokoi and Ohba developed a tool, called TCG, that generates tests for a cause-effect graph.[12] Based on a selected set of nodes in a cause-effect graph G, the set of all possible combinations of input conditions of G is divided into equivalence classes, one for each possible combination of the outcomes of the selected nodes. For example, if only an effect node E of G is selected, two combinations of input conditions are chosen, one making node E true and the other making node E false. In contrast, if all nodes of G are chosen, all combinations of input conditions are chosen. Now we show the application of this strategy to generate tests for C#, according to the syntax tree in Figure 2. Assume that we select nodes N_4 and N_5 for equivalence partitioning. N_4 and N_5 have three combinations of outcomes since it is impossible to make N_4 true and N_5 false at the same time. TCG chooses the constraint set $\{t_{13}, t_{14}, t_{15}\}$ for C# shown in Table 4. t_{13} makes both N_4 and N_5 true, t_{14} makes N_4 false and N_5 true, and t_{15} makes both N_4 and N_5 false. The constraint set $\{(t,t,f), (f,f,t), (f,f,f)\}$ does not distinguish C# from $((E_1<E_2)\ |\ (E_3>=E_4))\ |$

($E_5 = E_6$), which differs from C# in one binary operator only.

The BOR testing strategy

As shown earlier, one problem in testing compound predicates is the detection of Boolean operator faults. The Boolean operator (or BOR) testing criterion for a compound predicate is to guarantee the detection of (single or multiple) Boolean operator faults, including incorrect AND/OR operators and missing or extra NOT operators. A test set T for a predicate C is said to be a BOR test set for C if T satisfies the BOR testing criterion for C. A set S of constraints for predicate C is said to be a BOR constraint set for C provided that if a test set T for C satisfies S, T is a BOR test set for C. An algorithm that generates a minimum BOR constraint set for a compound predicate was given[3] and is referred to as algorithm BOR_min in this paper.

Here we now show the application of algorithm BOR_min to C#. We first transform C# into its syntax tree, as shown in Figure 2. Then we visit the nodes in the syntax tree for C# from leaf nodes to the root node. (Note that Elmendorf's strategy does the opposite.) Each node N_i, $i>0$, in the syntax tree for C#, corresponds to a predicate $P(N_i)$ in C# and is associated with true and false constraint sets, denoted as $T(N_i)$ and $F(N_i)$ respectively, such that:

- $T(N_i)$ is a set of constraints producing the true value for $P(N_i)$.
- $F(N_i)$ is a set of constraints producing the false value for $P(N_i)$.

When we visit each of nodes N_1, N_2, and N_3, we define its true constraint set as {(t)} and its false constraint set as {(f)}. For node N_4, we define $T(N_4)$ as {(t,t)} and $F(N_4)$ as {(t,f), (f,t)}. For node N_5, we construct its constraint sets according to the following rules:

$$F(N_5) = F(N_4) \,\%\, F(N_3) \text{ and}$$

$$T(N_5) = (T(N_4) \times \{f_3\}) \cup (\{f_4\} \times T(N_3)),$$

where "%" denotes the onto operation, "×" denotes the concatenation operation, f_3 is in $F(N_3)$, f_4 is in $F(N_4)$, and (f_4,f_3) is in $F(N_5)$.

Table 4 Constraint set $\{t_{13}, t_{14}, t_{15}\}$

	$((E_1<E_2)$ & $(E_3>=E_4))$	| $(E_5=E_6)$	outcome of C#
t_{13}	t	f	t
t_{14}	f	t	t
t_{15}	f	f	f

Table 5 Constraint set $\{t_{16}, t_{17}, t_{18}, t_{19}\}$

	$((E_1<E_2)$ & $(E_3>=E_4))$	| $(E_5=E_6)$	outcome of C#
t_{16}	t	f	t
t_{17}	t	t	t
t_{18}	f	f	f
t_{19}	t	f	f

$F(N_4)\%F(N_3)$ returns a minimum subset of the product of $F(N_4)$ and $F(N_3)$ such that each element in $F(N_4)$ or $F(N_3)$ is chosen at least once.[13] Since $F(N_4)$={(t,f), (f,t)} and $F(N_3)$={(f)}, $F(N_4)\%F(N_3)$ returns {(t,f,f), (f,t,f)}. f_3 must be {(f)}, and f_4 has two choices: {(t,f)} and {(f,t)}. By letting f_4 be {(t,f)}, $T(N_5)$ = {(t,t,f), (t,f,t)}. Therefore, algorithm BOR_min generates the constraint set $\{t_{16}, t_{17}, t_{18}, t_{19}\}$ for C# given in Table 5.

The constraint set {(t,t,f), (t,f,t), (t,f,f), (f,t,f)} is a minimum BOR constraint set for C#. The sizes of the constraint sets generated for C# by Elmendorf's strategy, equivalence partitioning testing, and algorithm BOR_min are 7, 3, and 4, respectively.

Earlier we showed two rules for the construction of true and false constraint sets for node N_5 in Figure 2. These two rules are used for an OR node. Similar rules are used for an AND node. The idea behind algorithm BOR_min is to derive a minimum constraint set to solve the problem of fault propagation, which is the propagation of an incorrect outcome of a portion of a compound predicate to an incorrect outcome of the compound predicate. For a predicate with $n>0$ AND/OR operators, algorithm BOR_min generates a minimum BOR constraint set, which contains at most $n+2$ constraints. Elmendorf's strategy also generates a BOR constraint set for a compound predicate, but the size of the generated constraint set is $(n+2)$ or more, up to $O(2^{**}n)$.[7] In the following discussion, BOR testing (or the BOR testing strategy) refers to the use of algorithm BOR_min to generate a constraint set for a compound predicate.

A constraint for a predicate is said to be infeasible for the predicate if it can never be covered by any

Table 6 Average fault detection rates

	S3	S4	S5
BOR testing	99.3%	99.7%	99.9%
Branch testing	72.2%	72.5%	72.9%

test for this predicate. For example, the constraint (t,t) is infeasible for predicate $((E_1>E_2) \mid (E_1=E_2))$, since the value of E_1 can never be both greater than and equal to that of E_2 at the same time. For $((E_1<E_2)\ \&\ (E_3>=E_4)) \mid (E_1<E_2)$, the constraint (t,t,f) is infeasible since it has two distinct values for $(E_1<E_2)$. If the constraint set produced by BOR testing for a predicate contains some infeasible constraints, then 100 percent coverage of the constraint set is impossible. The problem of infeasible constraints also exists in Elmendorf's strategy.

A comparison between BOR and branch testing

For a predicate with $n>0$ AND/OR operators, branch testing requires two tests, and BOR testing at most $(n+2)$ tests. We conducted an empirical study to compare the effectiveness of these two predicate testing strategies. Let a singular Boolean expression (SBE) be a Boolean expression in which each Boolean variable occurs only once. The reason for using SBEs is that we want 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.

We constructed the following sets of SBEs:

S3–A set of 48 mutually nonequivalent SBEs with three variables
S4–A set of 366 mutually nonequivalent SBEs with four variables
S5–A set of 2624 mutually nonequivalent SBEs with five variables

In each of S3, S4, and S5, these SBEs differ from one another in Boolean operators or parentheses or both. For each Boolean expression B in S3, we applied algorithm BOR_min to generate a test set T(B) and determined the fault detection rate of T(B), which is defined as

D(B) / (the number of tests in S3)

where D(B) is the number of Boolean expressions in S3 that can be distinguished from B by T(B).

Then we computed the average of these fault detection rates. This average value is referred to as the average fault detection rate of S3 using BOR testing. We also computed the average fault detection rates of S4 and S5 using BOR testing, as well as the average fault detection rates of S3, S4, and S5 using branch testing. (For branch testing of a Boolean expression, we chose two tests to satisfy complete branch testing.) Table 6 shows these average fault detection rates.

Our results show that BOR testing is more effective than branch testing for fault detection and that BOR testing almost guarantees the detection of Boolean operator faults and incorrect parentheses in a compound predicate.

A comparison of BOR testing, Elmendorf's strategy, and equivalence partitioning testing

We carried out an experiment to compare BOR testing with Elmendorf's strategy and equivalence partitioning testing. For a predicate C with $n>0$ AND/OR operators, BOR testing generates a minimum BOR constraint set with $(n+2)$ or fewer constraints, and Elmendorf's strategy generates a BOR constraint set with $(n+2)$ or more, up to $O(2**n)$, constraints. The number of constraints required for equivalence partitioning testing of C depends upon the selection of nodes in the syntax tree for C. In our empirical study, we selected all nodes in the syntax tree of C that denote Boolean operators. By doing so, equivalence partitioning testing generates $(n+1)$ or more, up to $O(2**n)$, constraints.[7]

We constructed a set, called SBE_4, of 51 nonequivalent SBEs with four Boolean variables. The SBEs in SBE_4 differ from one another in Boolean operators, parentheses, and/or the positions of Boolean variables. For each Boolean expression B in SBE_4, we (1) applied algorithm BOR_min to generate a test set T(B), and (2) determined the size of T(B) and the fault detection rate of T(B).

Then we computed the average size and fault detection rate for SBE_4. We also applied Elmendorf's strategy and equivalence partitioning testing to SBE_4 in a similar way. The average sizes and fault detection rates based on these three strategies are given in Table 7. The results show that

1. Elmendorf's strategy is very effective for detecting Boolean operator faults, incorrect parentheses, and interchanges of Boolean variables. (Note that the fault detection rate of a test set generated by Elmendorf's strategy for a compound predicate is not always 100 percent.)
2. BOR testing is almost as effective as Elmendorf's strategy and is slightly more effective than equivalence partitioning testing.
3. BOR testing requires about the same number of tests as equivalence partitioning testing and about half the number of tests as Elmendorf's strategy.

Statement 3 above is no longer true when the number of AND/OR operators in a compound predicate is larger than three. As mentioned earlier, for a predicate with $n>0$ AND/OR operators, the number of constraints generated by Elmendorf's strategy or equivalence partitioning testing is an *exponential function* of n, whereas the number of constraints generated by BOR testing is a *linear function* of n. Thus, as the number of AND/OR operators increases, BOR testing generates fewer constraints than equivalence partitioning testing. Another problem with equivalence partitioning testing is that it first generates all combinations of input conditions and then divides them into equivalence classes. BOR testing and Elmendorf's strategy do not have this problem, but they may generate infeasible constraints. In our opinion, BOR testing is more practical than the other two strategies.

Applying BOR testing to a boiler control and monitoring system

This section describes an application of BOR testing to the software for a simplified real-time boiler control and monitoring system. The specifications for the system 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. A version of the boiler control and monitoring system was developed at North Carolina State University.[14] This software system, which contains about 4500 lines of C code, was used in our empirical study. A brief description of the boiler system is given below. The objective and the details of this study are provided in the remainder of this section.

Table 7 Average sizes and fault defection rates

	Avg. Size	Avg. Fault Detection Rate
BOR testing	4.9	99.73%
Elmendorf's strategy	9.2	100.00%
Equivalence partitioning	4.5	96.88%

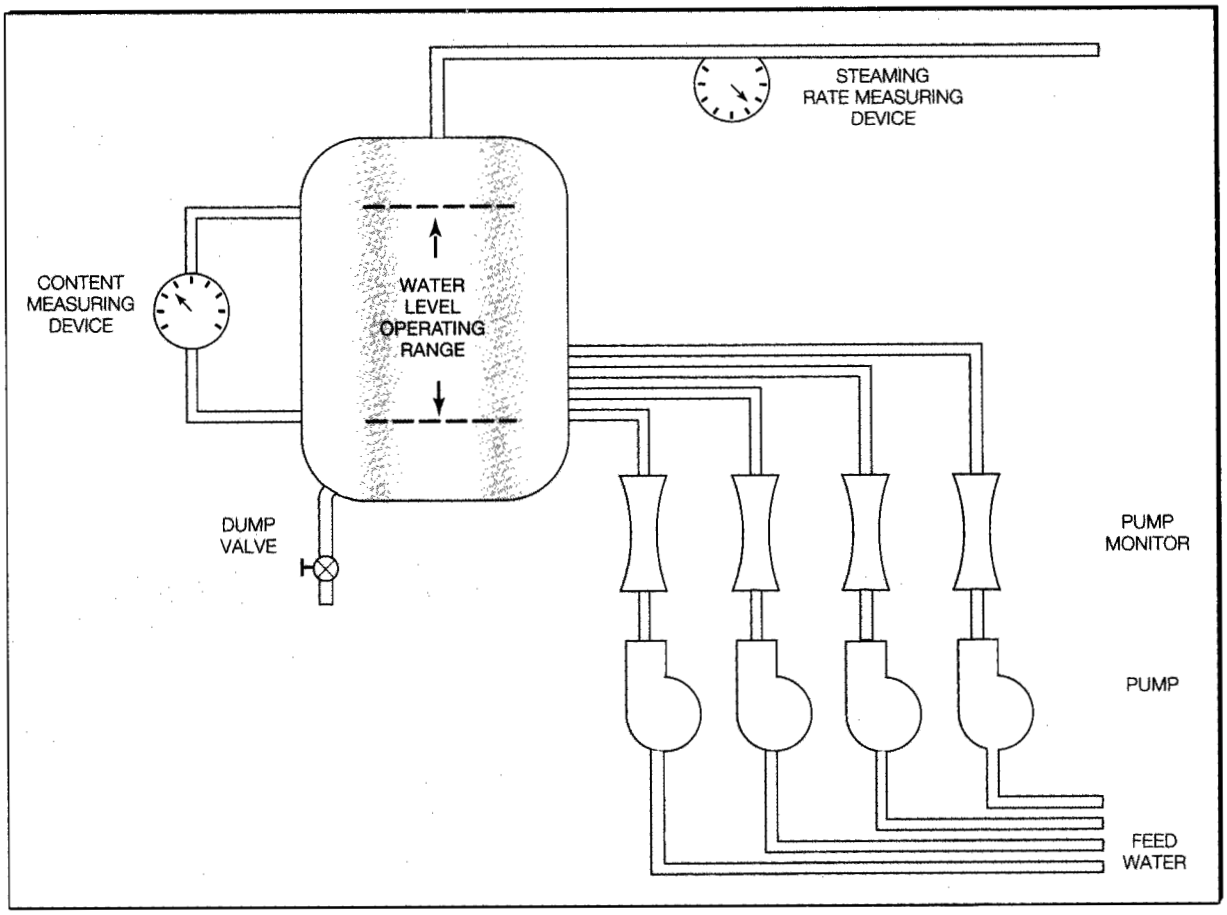
Figure 3 shows the context diagram for the simplified boiler control and monitoring system. This boiler system consists of a natural-gas-fired water-tube boiler producing saturated steam. The steam flow may vary rapidly and irregularly between zero and maximum, following a varying external demand. The water level in the boiler is regulated by the control of the inflow of feedwater. The water level must be kept between an upper and lower limit. 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 that 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:

1. To regulate the water level by controlling the inflow of feedwater by appropriately turning pumps on or off at required instances
2. To diagnose and isolate all potential errors and issue a correction or repair request when errors are discovered
3. To display at all times "best estimates" of various readings for the boiler operator
4. To accept appropriate operator commands

Objective and procedures of the boiler system study. During the development of the boiler system at North Carolina State University, the orig-

Figure 3 Context diagram for the boiler control and monitoring system



inal, informal specification of the system was rewritten in terms of a number of extended finite-state machines (EFSMs).[15] According to the boiler's EFSM specification, test suites for the unit, integration, and system testing of the boiler system were constructed to ensure thorough testing.[16] In addition to the coverage of every state and branch of individual EFSMs for the boiler system, 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 EFSMs according to the predicates in these EFSMs.

The objective of our study was to evaluate the EFSM specification-based test suites for the boiler system against the BOR testing criterion. We performed both specification- and program-based BOR testing of the boiler system as follows:

- For specification-based BOR testing of the boiler system, we chose the most critical effect, the "boiler shutdown" effect, in the boiler system and derived a cause-effect graph (CEG) for the shutdown effect (next subsection). From the EFSM-based test cases developed previously for the boiler system, we selected those related to the shutdown effect. The selected test set, referred to as the shutdown test set, contains 372 test cases. We used the shutdown test set to measure the BOR coverage of the CEG for the shutdown effect (shown later in this paper).
- For program-based BOR testing of the boiler system, we chose a module dealing with the

shutdown effect.[17] We measured the BOR coverage of this module by the shutdown test set (discussed later).

Derivation of a cause-effect graph for the shutdown effect. The CEG for the shutdown effect, referred to as the shutdown CEG, is organized in five levels. The level 1 (the highest level) CEG for boiler shutdown is shown in Figure 4. The annotations for nodes in the level 1 CEG follow:

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

The cause nodes of the level 1 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.

Measurement of BOR coverage of the shutdown cause-effect graph. When we attempted to measure the BOR coverage of the shutdown CEG by the shutdown test set, we encountered a problem. Although algorithm BOR_min generates a minimum BOR constraint set for a compound predicate, such a minimum BOR constraint set is not unique. As a result, the selection of a minimum BOR constraint set for a compound predicate may affect the BOR coverage of the predicate by a given test set. To solve this problem, an algorithm called BOR_cov was developed. For a given test set T for a predicate C, algorithm BOR_cov

1. Identifies the set T′ of redundant tests in T, which do not improve the capability of detection of Boolean operator faults in C
2. Produces a minimum set T″ of additional constraints that are needed for BOR testing of C
3. Computes the BOR coverage of C by T, which is defined as $(|T| - |T'|) / (|T| - |T'| + |T''|)$, where $|S|$ denotes the size of a set S.

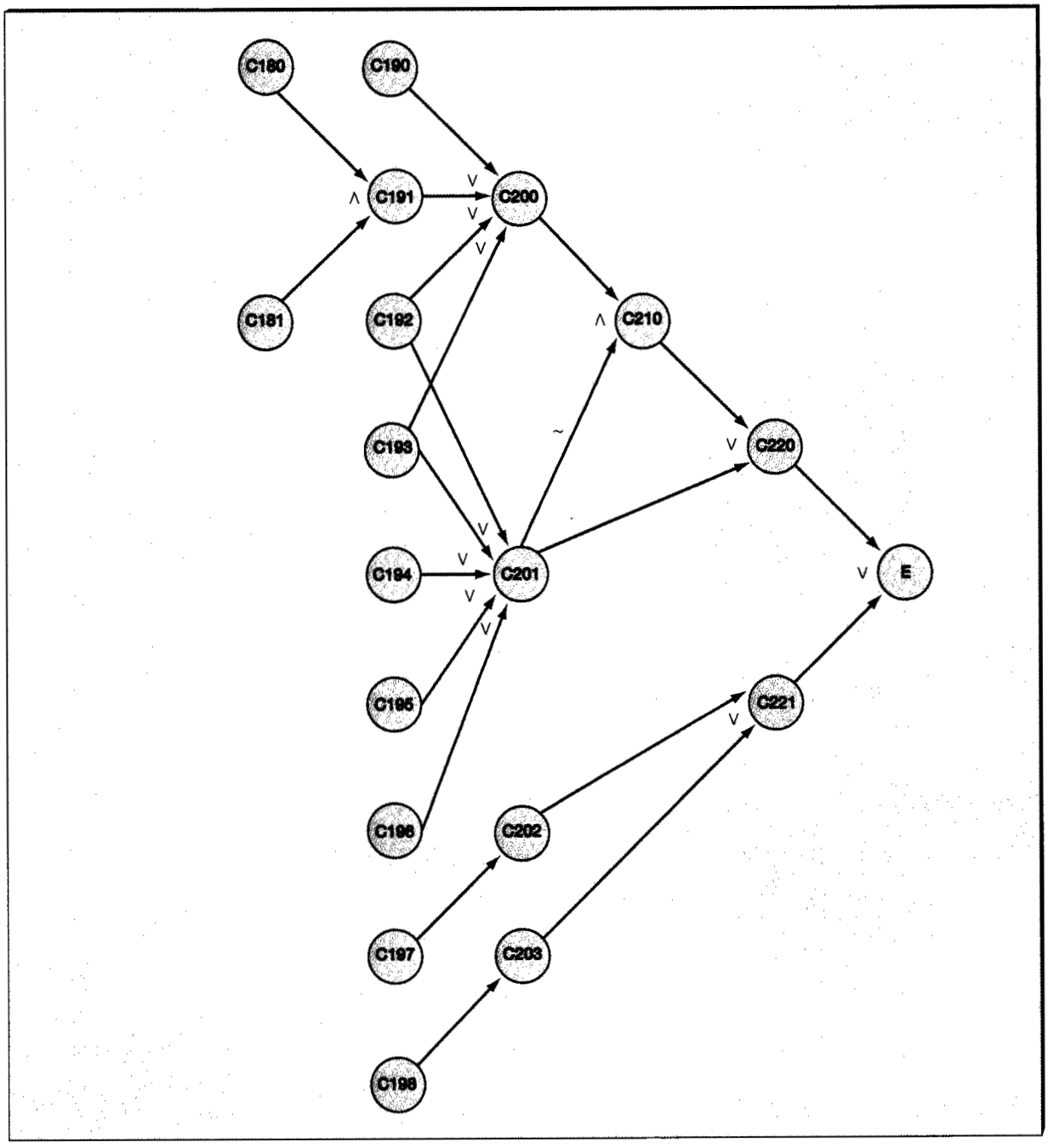
We used algorithm BOR_cov to measure the BOR coverage of the shutdown CEG by the shutdown test set. Of the 372 tests in the shutdown test set, 59 tests (about 1/6 of the total) were found to be redundant. Also, 24 more constraints are needed for BOR testing. So the BOR coverage of the shutdown CEG by the shutdown test set is $(372-59)\div(372-59+24) = 0.928$. Most of the redundant tests deal with pump and flow monitor combinations. However, most of the additional tests needed for BOR testing also deal with pump and flow monitor combinations. The reason is that the tests in the shutdown test set for combinations of pumps and flow monitors were selected without applying any effective predicate-based testing strategy.

The shutdown test set was constructed earlier by three persons totaling approximately 100 person-hours. In our study, the shutdown CEG was constructed by one person in about 20 hours. CEG-based test generation can be automated. Also, CEGs can be analyzed for the detection of ambiguities and inconsistencies in system specification.[1] Thus, the use of CEGs for software specification and test generation has significant advantages.

Measurement of BOR coverage of a module in the implementation of the boiler. As mentioned earlier, we chose one module in the implementation of the boiler to measure its BOR coverage. The selected module deals with the shutdown effect. It contains 360 statements in C and 34 predicates, of which 21 are simple predicates (that is, predicates without 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 in this module. (Two constraints are generated for each simple predicate and three constraints for each compound predicate with one AND/OR operator.)

The shutdown test set was used to execute the implementation of the boiler.[18] Based on the BOR

Figure 4 Level 1 cause-effect graph for the boiler control and monitoring system



coverage information collected from the selected module, 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 we investigated the two uncovered constraints, we discovered a "bug" in the selected module. This bug would have been discovered if the selected module had

been tested with 100 percent BOR coverage. Also, the two discovered constraints correspond to some of the additional tests needed for BOR testing of the shutdown CEG. Therefore, had these additional tests been used to execute the implementation of the boiler, the two uncovered constraints would have been covered and the bug discovered.

Applying BOR testing to a set of *N*-version programs

This section reports an empirical study of applying BOR testing to a set of *N*-version programs written in Pascal.[19] Our reason for using *N*-version programs is that multiple functionally equivalent programs provide more objective results than just one program. We chose five functionally equivalent Pascal programs, which were produced as part of another study. These five programs were written independently by graduate students to solve a navigational problem that was an extension of the earth satellite problem.[20] The sizes of these five programs range from 400 to 800 Pascal statements.

Acceptance testing of these five programs involved both random and functional testing and used a tool called BGG, which was developed at North Carolina State University to measure the test coverage of statements, branches, and various types of data flow metrics for Pascal programs.[21] A set of 1000 random tests was generated by using a uniform distribution of all input values. A set of 103 functional tests[22] was generated by considering extreme and special values such as singularities and boundaries. The results of acceptance testing of these programs were reported.[23,24] BGG was recently extended to generate BOR constraint sets for predicates in a Pascal program and to measure the coverage of these BOR constraints according to a given test set for the program.

The objective of this empirical study was to use the five-version Pascal programs to compare BOR testing with random and functional testing. According to the specification for the navigational problem, we derived a CEG for the specification and applied algorithm BOR_min to generate a set of 43 test cases, referred to as the CEG-BOR test set for the five-version programs. We applied the CEG-BOR test set, using the BGG tool, to execute each of the five-version programs, and computed the average coverages of statements and branches, respectively, of the five-version programs. We applied the random and functional test sets for the five-version programs in a similar way. Table 8 shows these average coverages.

Table 8 Average coverages for test sets

	Statement Coverage	Branch Coverage
Random	0.848	0.634
Functional	0.960	0.896
CEG-BOR	0.963	0.896

From Table 8, the CEG-BOR test set provides about the same statement or branch coverage as the functional test set. However, the size of the CEG-BOR test set is about 40 percent of that of the functional test set.

Since the five-version programs contain faults, we also compared the CEG-BOR, random, and functional test sets for their effectiveness of fault detection. The combination of random and functional test sets detected all faults in one of the five-version programs and detected all but one fault in each of the other four programs. The CEG-BOR test set detected all faults in each of the five-version programs. Thus, in this experiment, the CEG-BOR test set is more effective than the combination of random and functional test sets. As an example, for one of the five-version programs, the random test set detected three faults, the functional test set five faults, and the CEG-BOR test set all nine faults in the program.

Summary

In this paper we have presented the results of four empirical studies of the BOR testing strategy. Two of these studies involved the use of Boolean expressions and the other two the use of actual programs. The major findings are the following:

- BOR testing is effective for detecting faults in a compound predicate.
- BOR testing is more cost-effective than several other predicate testing strategies.
- BOR testing based on a cause-effect graph representing software specification is practical and effective for detecting faults in the corresponding implementation.

The use of cause-effect graphs in software specification and design is not yet popular. One possible reason is the lack of user-friendly tools for expressing software specification and design in cause-effect graphs. Since CASE (computer-aided software engineering) tools supporting cause-effect graphs are becoming available,[12,25] the use of cause-effect graphs will increase.

Two variations of the algorithm BOR_min were studied.[26] Two extensions of the BOR testing criterion were proposed,[3,27] one to include the detection of incorrect relational operators and the other to include the detection of incorrect relational operators and arithmetic expressions. We will investigate other extensions of BOR testing such as nonarithmetic operations (for example, comparison between two pointers or character strings) and additional Boolean operators (for example, exclusive-OR and short-circuit AND/OR). Also, we plan to carry out empirical studies of BOR testing by using larger software systems, and we plan to investigate the implementation of tools to support specification- and program-based BOR testing.

Acknowledgment

The authors would like to thank H. K. Su for his earlier work on the comparison between BOR testing and Elmendorf's strategy.

Cited references and notes

1. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., New York (1979).
2. B. Beizer, *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold Co., Inc., New York (1990).
3. K. C. Tai, "Predicate-Based Test Generation for Computer Programs," *Proceedings of the International Conference on Software Engineering* (May 1993), pp. 267–276.
4. This research was supported in part by the IBM Centre for Advanced Studies, NASA Grant NAG-1-983 and NSF Grant CCR-8907807.
5. "Testing criterion" and "testing strategy" are often used as synonyms. In this paper, we distinguish the two only when a testing criterion can be satisfied by using different testing strategies.
6. W. R. Elmendorf, *Cause-Effect Graphs on Functional Testing*, TR-00.2487, IBM Systems Development Division, Poughkeepsie, NY (1973).
7. K. C. Tai, *Structure- and Fault-Based Testing Strategies for Compound Predicates*, TR-94-05, Department of Computer Science, North Carolina State University, Raleigh, NC (1994).
8. For languages like C that use short-circuit evaluation of AND/OR operators, more than two tests are needed.
9. K. A. Foster, "Error Sensitive Test Cases Analysis (ESTCA)," *IEEE Transactions on Software Engineering* **SE-6**, No. 3, 258–264 (May 1980).
10. W. E. Howden, "Weak Mutation Testing and Completeness of Test Cases," *IEEE Transactions on Software Engineering* **SE-8**, No. 4, 371–379 (July 1982).
11. The syntax tree of a predicate can be viewed as a cause-effect graph with exactly one effect node and with a cause node being a relational expression or Boolean variable.
12. S. Yokoi and M. Ohba, "TCG: CEG-Based Tool and Its Experiments," *Proceedings of the 13th Software Reliability Symposium*, Nara, Japan (November 1992), pp. 41–49.
13. If the "%" operation has two or more possible values, it returns just one of them.
14. M. A. Vouk and A. Paradkar, "Design and Review of Software Controlled Safety-Related Systems: The NCSU Experience with the Generic Problem Exercise," *Proceedings of the International Invitational Workshop on the Design and Review of Software Controlled Safety-Related Systems*, Ottawa (June 1993).
15. An extended finite-state machine is an FSM with the use of variables and predicates.
16. A. Paradkar, I. Shields, and J. Waters, *The NCSU Solution to the Generic Problem Exercise: Boiler Control and Monitoring System*, Department of Computer Science, North Carolina State University, Raleigh, NC (May 1993).
17. The selected module is not the only module dealing with the shutdown effect. We chose only one module since no automatic tool was available for measuring the BOR coverage of C programs.
18. Since the selected module is one of several modules dealing with the shutdown effect, only a portion of the shutdown test set invokes the selected module.
19. *N*-version programming is a technique for increasing software reliability; it requires independent development of multiple versions of a software system for a given specification and then execution of these versions at the same time to compare their results.
20. P. M. Nagel and J. A. Skrivan, *Software Reliability: Repetitive Run Experimentation and Modelling*, BSC-40336, Boeing Corporation, Seattle, WA (1982).
21. M. A. Vouk and R. E. Coyle, "BGG: A Testing Coverage Tool," *Proceeedings of the 7th Northwest Software Quality Conference* (1989), pp. 212–233.
22. W. E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill Book Co., Inc., New York (1987).
23. M. A. Vouk, D. F. McAllister, and K. C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software," *Proceedings of the Workshop on Software Testing* (July 1986), pp. 74–81.
24. M. A. Vouk, M. L. Helsabeck, D. F. McAllister, and K. C. Tai, "On Testing of Functionally Equivalent Components of Fault-Tolerant Software," *Proceedings of COMPSAC (Computer Software and Applications) '86* (October 1986), pp. 414–419.
25. *Software Through Pictures*, Interactive Development Environments, Inc., San Francisco, CA.
26. K. C. Tai and H. K. Su, "Test Generation for Boolean Expressions," *Proceedings of COMPSAC (Computer Software and Applications) '87* (1987), pp. 278–283.
27. K. C. Tai, *A Theory of Fault-Based Predicate Testing for Computer Programs*, TR-94-04, Department of Computer Science, North Carolina State University, Raleigh, NC (1994).

Accepted for publication April 12, 1994.

Kuo-Chung Tai *Computer Science Department, North Carolina State University, Raleigh, North Carolina 27695-8206 (electronic mail: kct@csc.ncsu.edu).* Dr. Tai is a professor of computer science at North Carolina State University. He has published papers in the areas of software engineering, distributed systems, programming languages, and compiler construction. His current research interests include software testing, concurrent programming languages, and analysis, testing, and debugging of concurrent software and communication protocols. Dr. Tai received his Ph.D. degree in computer science from Cornell University. From 1989 to 1991, he was the director of the Software Engineering Program at the National Science Foundation. He is an associate editor of *Computer Languages, International Journal of Software Engineering and Knowledge Engineering,* and *International Journal of Computer and Software Engineering.* He is a co-program chair of the 1994 International Conference on Parallel Processing.

Mladen A. Vouk *Computer Science Department, North Carolina State University, Raleigh, North Carolina 27695-8206 (electronic mail: mav@csc.ncsu.edu).* Dr. Vouk received his Ph.D. degree from the University of London (UK). He has extensive experience in both commercial software production and academic computing environments. He is the author, or coauthor, of over 80 publications. He is currently an associate professor of computer science at North Carolina State University. His research and development interests include: software process modeling and risk management; software testing, reliability, and fault-tolerance; development of large-scale scientific software-based systems; and high-speed networking issues. He is the chairman-elect of the IFIP Working Group 2.5 on Numerical Software, and the cochairman of the Software Quality Interest Sub-Committee of the North Carolina Quality Assurance Discussion Group. He is an associate editor of *IEEE Transactions on Reliability.*

Amit M. Paradkar *Computer Science Department, North Carolina State University, Raleigh, North Carolina 27695-8206 (electronic mail: amit@bvcd.ncsu.edu).* Mr. Paradkar received his M.S. degree in computer studies from North Carolina State University and is currently a Ph.D. student in the Computer Science Department at NCSU. He was a system analyst with Citicorp Overseas Software Limited in Bombay, India. His research interest includes software fault-tolerance, software testing, software reliability, and software process modeling.

Peng Lu *IBM Software Solutions Division, Toronto Laboratory, 844 Don Mills Road, North York, Ontario M3C 1V7, Canada.* Dr. Lu is a research staff member at the IBM Centre for Advanced Studies (CAS) of the Toronto laboratory. He is currently a principal investigator for the Software Reliability and Testing project under CAS. Dr. Lu received his Ph.D. in engineering from McMaster University in 1989 and joined the IBM Toronto Laboratory in 1990. He has worked on several development projects and on new technology transfer, including CASE, process modeling, reverse engineering, and software reliability engineering. His research interests include software reliability and testing, software engineering, artificial intelligence, and expert systems.

Reprint Order No. G321-5550.