# Specification-based testing using cause-effect graphs [*]

Amit Paradkar [**], K.C. Tai and M.A. Vouk

*Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA*
E-mail: {kct,vouk}@adm.csc.ncsu.edu

In this paper we discuss the advantages and limitations of a specification-based software testing technique we call CEG-BOR. There are two phases in this approach. First, informal software specifications are converted into cause-effect graphs (CEG). Then, the Boolean OperatoR (BOR) strategy is applied to design and select test cases. The conversion of an informal specification into a CEG helps detect ambiguities and inconsistencies in the specification and sets the stage for design of test cases. The number of test cases needed to satisfy the BOR strategy grows linearly with the number of Boolean operators in CEG, and BOR testing guarantees detection of certain classes of Boolean operator faults. But, what makes the approach especially attractive is that the BOR based test suites appear to be very effective in detecting other fault types. We have empirically evaluated this broader aspect of the CEG-BOR strategy on a simplified safety-related real-time control system, a set of N-version programs, and on elements of a commercial data-base system. In all cases, CEG-BOR testing required fewer test cases than those generated for the applications without the use of CEG-BOR. Furthermore, in all cases CEG-BOR testing detected all faults that the original, and independently generated, application test-suites did. In two instances CEG-BOR testing uncovered additional faults. Our results indicate that the CEG-BOR strategy is practical, scalable, and effective across diverse applications. We believe that it is a cost-effective methodology for the development of systematic specification-based software test-suites.

## 1. Introduction

As software becomes more complex and more infused into every aspect of daily life, it becomes increasingly more important to provide assurances that software is trustworthy [Hamlet 1995]. Formal specification and design of a software system, accompanied by formal verification and validation, would be an ideal vehicle for assuring software quality. However, in practice the prevalent approach is through testing. The problem is that testing is often *ad hoc* and the gains in software quality may not be commensurate with the efforts. This is a source of continuous concern for both software industry and software customers. Therefore, it is not surprising that the field of software testing is a very active area of research (see recent surveys in [Hamlet 1995; White 1995]).

[*] This work was supported in part by NSF grant CCR-9320992, MCNC/EPA grant C94-7050-027/029, NASA grant NAG-1-983, and the Center for Advanced Studies, IBM Canada.

[**] Author's present address: T.J. Watson Research Center, IBM Corp., 30 Saw Mill River Road, Hawthorne, NY 10532, USA. E-mail: amit@watson.ibm.com.

The most common approach to software testing, black-box testing, is specification based. Typically, a tester studies software specification documents (both requirements and design) and develops a set of test cases that he/she believes cover the principal product functionality and problem areas. Unfortunately, in practice most of the software specifications are stated informally and that leaves a lot of room for ambiguities and misinterpretations. *Systematic* specification-based strategies include boundary and special value testing, state-based testing, cause-effect graph testing, functional testing, random testing and testing by equivalence partitioning [Howden 1987; Myers 1979; Beizer 1990]. But, even systematic specification-based testing strategies may yield unsatisfactory results because they may not be scalable or consistently effective across a wide range of applications.

Less frequently, software practitioners also develop test cases based on the software structure. This approach is often called white-box (also glass-box or clear-box) testing. A concept that is often associated with white-box testing is code (or structure) coverage. Testing is considered *complete* or *adequate* under a given metric if the executed test cases exercise (cover) at least once a pre-determined fraction of the metric structures that exist in the program. Structural and program-based strategies include statement and branch testing, data-flow testing, domain testing, mutation testing, and so on [Howden 1987; Beizer 1990].

There are many practical problems with white-box methodologies. For example, often neither testers nor users relate to any but the most trivial structure-based metrics that guide the white-box testing. This erodes the confidence in the value of the strategy and makes the test-case generation counter-intuitive. Also, for many white-box metrics, it is still unclear what, if anything, 90% (or even 100%) coverage of the metric guarantees in the way of fault detection or software performance in the field. Furthermore, the number of test cases required to satisfy some of the coverage criteria can be inordinately large and this usually poses a significant problem.

In general, one of the major obstacles to practical use of many testing methodologies is that they do not simultaneously possess the properties of effectiveness, application range, performance robustness, and scalability. By effectiveness we mean the ability of the test sets generated by a particular strategy to detect a wide spectrum of faults. By range of the methodology, we mean its ability to provide the same fault-detection performance across a range of different application areas. By performance robustness, we mean the ability of the methodology to repeatedly and consistently perform at some predetermined level of effectiveness under a variety of conditions. For example, if a methodology requires 100% branch coverage, does it guarantee detection of a certain class or fraction of faults? By scalability we mean the rate at which the number of test cases the methodology requires increases as the size/complexity of the software increases. For example, a methodology that generates effective test cases for small programs may not scale to large software because the number of test cases grows exponentially with a linear increase in the size of the software. Such a testing methodology may not be practical.

Although some specification-based testing techniques have been shown to possess better fault detecting properties than some structure-based testing approaches [Basili and Selby 1987; Foreman and Zweben 1993], an acceptable and practical methodology must combine elements of both black-box and white-box testing and must build on the strengths but avoid the problems associated with either approach. We believe that structured specification-based testing represents a step in that direction. Like Laski [1989], we advocate a methodology that formalizes the logic of the specifications and then applies structural techniques to design and implement test cases that can be used throughout the software life cycle. Laski proposes to use the specification predicates to a) develop a class of black-box test cases and b) to refine specification predicates in several stages so that the test cases can be generated incrementally (using control and data-flow techniques). However, the techniques for the actual test code generation are not fully specified [Laski 1989].

A methodology which we believe possesses effectiveness, range, robustness, and most of all scalability is an extension of the classical cause-effect graph approach. The particular specification-based testing methodology that we describe in this paper uses cause-effect graphs (CEG) in combination with the Boolean OperatoR (BOR) test strategy. We call it CEG-BOR. The CEG-BOR strategy is also based on propositional logic representation of the specification and it is designed to detect Boolean operator faults. In addition, it is also very effective in detecting other types of faults.

Section 2 provides basic definitions and necessary background information. In section 3, we discuss some issues involved in cause-effect graph based specification and testing and describe the CEG-BOR test generation. In section 4 we discuss application of CEG-BOR strategy to a real-time safety-related application, a set of N-version programs, and to elements of a commercial data base system. In section 5 we discuss limitations of the CEG-BOR strategy and suggest extensions that overcome some of these limitations. Conclusions are given in section 6.

## 2. Definitions and notation

A *cause-effect graph* [Myers 1979; Elmendorf 1973; IEEE-ANSI 1989] is a graphical notation for describing logical relationships between causes and effects in a software specification. A cause is a binary-valued atomic sentence in the specification and represents an input value (or state or event), an effect is also a binary-valued atomic sentence representing an output state or action. For each feature in the software specifications, we isolate causes that influence the feature's behavior. Then, we derive a graphical representation of the cause and effect relationships by connecting the causes and effects with Boolean operators.

For example, consider the following natural language specification: "The boiler should be shut down if any of the following conditions occurs:

1. If the water level in a boiler is below the 20,000 lb. mark, or
2. the water level is above the 120,000 lb. mark, or

3. the boiler is operating in a degraded mode and the steam meter fails.

The boiler is in a degraded mode if either a water pump or a pump monitor fails." This *compound* sentence is composed of six *atomic* sentences:

1. The water level in a boiler is below the 20,000 lb. mark. ($a$)
2. The water level is above the 120,000 lb. mark. ($b$)
3. A water pump has failed. ($c$)
4. A pump monitor has failed. ($d$)
5. Steam meter has failed. ($e$)
6. Shut the boiler down. ($f$)

Atomic sentences 1 through 5, which could be true or false, are the causes in the software specification. Certain combinations of these causes lead to the effect of the boiler being shut down. We associate a *Boolean variable* with each cause (as shown next to each of the six atomic sentences). Then Boolean variables $a$ through $e$ constitute the input causes in the graph. We can represent the desired effect $f$ by a Boolean expression $\mathcal{E} = a + b + (c + d)e$, i.e. effect $f$ is true if $\mathcal{E}$ is true. Graphical representation of this CEG is given in figure 1. Each cause/effect has a node associated with it. Nodes $a$ through $e$ are called *cause nodes* and node $f$ is called *effect node*. Note the *intermediate* nodes corresponding to $c + d$ and $(c + d)e$.

Formally, an *effect* in a *CEG* is defined as follows:

- A cause by itself implies an effect in a CEG.
- If $\mathcal{G}_1$ and $\mathcal{G}_2$ are effects in a CEG, then so are $(\overline{\mathcal{G}_1})$, $(\mathcal{G}_1 + \mathcal{G}_2)$, and $(\mathcal{G}_1 \cdot \mathcal{G}_2)$, where "$\cdot$", "$+$", and " $\overline{\phantom{x}}$ " are Boolean operators representing conjunction, disjunction, and negation respectively (the "$\cdot$" operator is usually omitted and is assumed to have higher precedence than the $+$ operator).

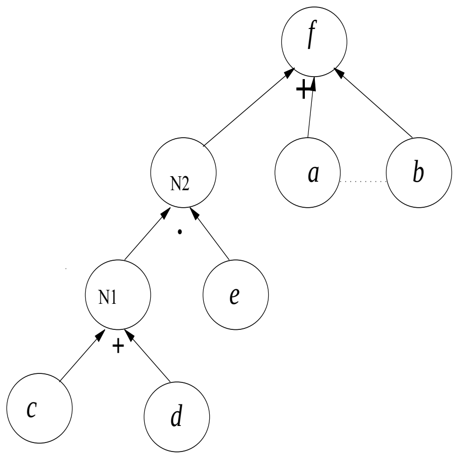

Figure 1. An example of a cause-effect graph.

Often, certain combinations of causes are not feasible. For example, in the CEG of figure 1, causes $a$ and $b$ are mutually exclusive. The CEG notation of [Elmendorf 1973] allows specification of such restrictions on combinations of causes. These restrictions can be defined by the user or in some cases can be implicitly derived by analyzing the causes. The mutual exclusion restriction between causes $a$ and $b$ is denoted by the dotted line between the two causes in the CEG of figure 1. With this restriction, the effect $f$ of figure 1 can also be described by the Boolean expression $(\overline{a} + \overline{b})(a + b + (c + d)e)$. In this case, the mutual exclusion restriction is built into the Boolean expression. It should be noted that the graphical notation for specifying restrictions does not warrant a change in the formal definition of an effect in a CEG since "$\cdot$", "$+$", and " $\overline{\phantom{x}}$ " are functionally complete operators. Since an effect is an implied Boolean function of the causes, we will use the terms CEG and Boolean expression synonymously in the remainder of this paper.

A *singular Boolean expression* is a Boolean expression where each constituent Boolean variable occurs exactly once. For example, $\mathcal{E}_1 = (a + b)(\overline{c}d)$ is a singular Boolean expression and $\mathcal{E}_2 = ab + \overline{a}c$ is a non-singular Boolean expression. For a Boolean expression $\mathcal{E}$ with $n - 1$ binary Boolean operators (where $n > 1$), a *test constraint* is defined as a vector $(v_1, \ldots, v_n)$, where $v_i$, $1 \leqslant i \leqslant n$, is a symbol specifying a restriction on the $i$th Boolean variable in $\mathcal{E}$. Note that the unary negation operator does not affect the number of elements in the test constraint. For a Boolean variable, say $b$, we use the following symbols to denote different types of restrictions on its value:

- $t$ – the value of $b$ is true,
- $f$ – the value of $b$ is false.

We define $\mathcal{E}(v)$ to be the *outcome* or *result* of $\mathcal{E}$ when test constraint $v$ is applied to it. $\mathcal{E}(v)$ is either true or false. If $\mathcal{E}(v)$ is true, it indicates the occurrence of the effect in the CEG corresponding to $\mathcal{E}$. Let $S(\mathcal{E})$ denote a set of test constraints for $\mathcal{E}$. Then, $S(\mathcal{E})$ can be partitioned into $S_t(\mathcal{E})$ and $S_f(\mathcal{E})$, where

$$\forall u \in S_t(\mathcal{E})\ \mathcal{E}(u) \text{ is true} \quad \text{and} \quad \forall v \in S_f(\mathcal{E})\ \mathcal{E}(v) \text{ is false.}$$

For the CEG in figure 1 (denoted by Boolean expression $a + b + (c + d)e$), $v = (t, f, f, f, t)$ is a test constraint (where the first symbol $t$ in $v$ implies that cause $a$ is true, second symbol $f$ in $v$ implies that cause $b$ is false, and so on) which leads to the boiler being shut down. A *test case* is an assignment to the input variables which satisfies a given test constraint. The test constraint $v$ is satisfied by assigning a value of 10000 lb. to the input variable *water level* and by forcing the steam meter to fail with all the water pumps and pump monitors staying operational. In this paper, we developed the test cases for the specifications under test manually using the test constraints generated for a CEG.

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

- Boolean operator fault, which includes:

– *operator replacement* fault, which refers to the replacement of one binary Boolean operator with another,
– *variable negation* fault, which refers to the missing of a negation operator for a literal (which is either a Boolean variable or negation of a Boolean variable),
– *expression negation* fault, which refers to the missing of a negation operator for an expression,

- *incorrect parenthesis* fault,
- *incorrect Boolean variable* fault,
- *extra/missing binary Boolean operator* fault.

An incorrect Boolean expression may contain either a *single fault* or *multiple faults* of the same or different type.

Let $u = (u_1, \ldots, u_m)$ and $w = (w_1, \ldots, w_n)$, where $m, n > 0$, be two vectors. We define the *concatenation* of $u$ and $w$, denoted by $(u, w)$, as $(u_1, \ldots, u_m, w_1, \ldots, w_n)$. Let $A$ and $B$ be two sets. $A \cup B$ denotes the *union* of $A$ and $B$, $A \times B$ their *product* which is a set of all ordered pairs $(a, b)$ such that $a \in A$ and $b \in B$, and $|A|$ the *size* (or *cardinality*) of $A$. We define $A \otimes B$, called *onto* from $A$ to $B$, to be a minimum set $C \subseteq A \times B$ such that

$$\bigl(\forall a \in A\ \bigl(\exists (c, d) \in C\bigr) \mid a = c\bigr) \quad \text{and} \quad \bigl(\forall b \in B\ \bigl(\exists (e, f) \in C\bigr) \mid b = f\bigr).$$

Thus $|A \otimes B|$ is max($|A|, |B|$). If both $A$ and $B$ have two or more elements, $A \otimes B$ has several possible values and returns any one of them. For example, assume that $A = \{(a), (b)\}$ and $B = \{(c), (d)\}$. Then, $A \otimes B$ has two possible sets of elements: $\{(a, c), (b, d)\}$ and $\{(a, d), (b, c)\}$.

The BOR testing strategy for a Boolean expression $\mathcal{E}$ requires a set of test constraints to *guarantee* the detection of Boolean operator faults in $\mathcal{E}$ [Tai 1996]. A test constraint set for $\mathcal{E}$ is said to be a *BOR constraint set* for $\mathcal{E}$, if this set satisfies the BOR testing strategy for $\mathcal{E}$. Tai [1996] presented an algorithm for generating a BOR constraint set for a Boolean expression and showed that such a BOR set is minimal, and is bounded above by $n + 2$ where $n$ is the number of binary Boolean operators in the Boolean expression.

## 3. Issues in CEG-based specification and testing

In considering the use of CEG for specification representation and test generation, one has to weigh several things. On the one hand, because of their graphical nature, CEG-based specifications are often easier to comprehend than specifications using other notations. Also, because CEG have strong underpinnings in propositional logic, CEG-based specifications may provide an adequately formal framework for reasoning about them as well as for the generation of test-cases. On the other hand, conversion of an informal specification into CEG may be tedious and time-consuming. This is probably a major reason for very limited use of CEG in practice. Another likely reason is that only very few commercial tools support CEG diagramming and test generation (e.g., [Bender 1991]), while research tools such as TCG [1992] would need substantial improvements before they could be used in production environments. Furthermore, the previous approaches to CEG-based test generation are either impractical or ineffective. In section 3.1 we provide suggestions for converting informal specification into CEG. In section 3.2 we discuss two previous approaches to CEG-based test generation. In section 3.3 we present a new CEG-based test generation algorithm.

### *3.1. Converting a specification into CEG*

We believe that the process of construction of specification CEG pays back through its ability to uncover ambiguities and inconsistencies in the specifications, and that it is worth the cost. Nevertheless, for a large software system, the derivation of a single CEG for the whole system is not practical. Instead, we suggest the following strategy for managing the CEG construction phase:

- First, identify major software specification effects using the Pareto principle [Boehm 1989; Konz 1995] and construct CEG only for these critical effects. In the context of a software system, Pareto rule states that only a relatively few software modules or functions are critical and perform most of the tasks in the system. The issue is how too rank effects by criticality. We recommend the use of approaches based on the theory of operational profiles [Lyu 1994] and the risk-management techniques [Boehm 1989]. Examples of criteria that could be used to identify and rank the major effects are: the frequency of usage – e.g., in a transaction processing system some types of transactions are performed far more frequently than others; the relative importance of the effect in the system – e.g., in a banking system, financial transactions tend to be more important than the inquiry transactions; the criticality of the effects in terms of risk to life; and so on.
- Second, for an effect that has a large CEG, identify major intermediate nodes, divide the complete CEG into component CEG, and incrementally construct the component CEG. For a CEG $\mathcal{G}$ identification of major intermediate nodes is based on analysis of causes and effects of $\mathcal{G}$. Based on the major intermediate nodes in $\mathcal{G}$, $\mathcal{G}$ can be divided into a number of smaller CEG, which are grouped into levels. The highest level CEG of $\mathcal{G}$ have the effects of $\mathcal{G}$ as effects and have the major intermediate nodes as causes. This refinement continues in an incremental manner until we reach the atomic causes at the lowest level. This improves the readability of the CEG-based specification and also assists in change management since modifications can be localized.

An example showing how to use this approach is given in subsection 4.1.

### *3.2. Test generation from a CEG*

Elmendorf [1973], who developed the CEG notation, gave a test constraint generation algorithm for a CEG. This algorithm was described in [Myers 1979]. Below we illustrate the application of this strategy by generating test constraints for Boolean expression $\mathcal{E} = ab + c$. Figure 2 shows the CEG corresponding to $\mathcal{E}$. It contains three cause nodes $a, b$, and $c$. Node N4 corresponds to the "$\cdot$" operator, and node N5 corresponds to the "$+$" operator. The nodes in the CEG for $\mathcal{E}$ are visited from the effect node to the cause nodes. For node N5, inputs $(t, f)$, $(f, t)$ and $(f, f)$ are selected, with the first element of each input being the output of N4, and the second element of each input being the value of $c$. For node N4 with output value "$t$" we select input $(t, t)$ and for node N4 with output value "$f$", we select, inputs $(t, f)$, $(f, t)$ and $(f, f)$, where the first element of each input is the value of "$a$" and the second element of each input is the value of "$b$". Thus, Elmendorf's strategy generates the seven (7) element test constraint set for $\mathcal{E}$ as shown in table 1. In general, the number of test constraints generated by this algorithm grows exponentially with the number of input causes. Obviously, this can represent a practical obstacle. Some other problems encountered with the original Elmendorf algorithm are discussed by Nursimulu and Probert [1995].

Yokoi and Ohba [1992] developed a tool called TCG to generate test constraints based on a CEG. Given a selected set of nodes in a CEG 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

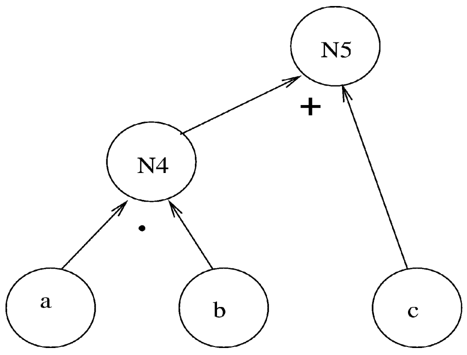

Figure 2. Cause-effect graph for $ab + c$.

Table 1
Elmendorf's test constraint set for CEG represented as $ab + c$.

| Test constraint $u$ | $\mathcal{E}(u)$ |
|---|---|
| $(t, t, f)$ | $t$ |
| $(t, f, t)$ | $t$ |
| $(f, t, t)$ | $t$ |
| $(f, f, t)$ | $t$ |
| $(t, f, f)$ | $f$ |
| $(f, t, f)$ | $f$ |
| $(f, f, f)$ | $f$ |

Table 2
TCG test constraint set for CEG represented as $ab + c$.

| Test constraint $u$ | Outcome for N4 | $\mathcal{E}(u)$ |
|---|---|---|
| $(t, t, f)$ | $t$ | $t$ |
| $(f, f, t)$ | $f$ | $t$ |
| $(f, f, f)$ | $f$ | $f$ |

one effect node, say E of G is selected, then two combinations of input conditions are chosen, one making node E true, and the other making node E false. On the other hand, if all nodes of G are chosen, then all combinations of input conditions are chosen. Consider application of this strategy to generate test constraints for the CEG shown in figure 2. Assume that we select nodes N4 and N5 for equivalence partitioning. All combinations of $a$ and $b$ that result in a particular truth value of N4 form one equivalence class. For example, $(t, f)$, $(f, t)$, and $(f, f)$ all make N4 false. Because N5 represents an "$+$" node, N4 and N5 have only three feasible combinations of outcomes (it is impossible to make N4 true and N5 false at the same time). Out of the available equivalent test constraints, TCG chooses the test constraint set shown in table 2.

The test constraint set in table 2 does not distinguish $\mathcal{E}$ from $(a + b) + c$, which differs from $\mathcal{E}$ in only one binary operator. Thus, the fault detection capability of the test constraints generated using this approach may not be acceptable. Moreover, the number of test constraints generated using this strategy can also be exponential.

## 3.3. *CEG-BOR test generation*

As mentioned earlier, an algorithm for generating a BOR constraint set for a Boolean expression was shown by Tai [1996]. This algorithm can be easily extended to generate a test constraint set for each node in a CEG. Figure 3 describes the CEG-BOR rules for generating a test constraint set for a CEG node. In the CEG-BOR strategy, the nodes of a CEG are visited from cause nodes to effect nodes. When a node is visited, a test constraint set for this node and its associated subgraph is constructed. After an effect node is visited, a test constraint set for the CEG associated with the effect node is available. We illustrate application of the CEG-BOR rules to the CEG (expression $\mathcal{E} = ab + c$) in figure 2. According to rule 1, the test constraint set for each of $a, b$, and $c$ is $\{(t), (f)\}$. For node N4, we apply rule 2 and get $\mathcal{CEGBOR}_t$(N4) $= \{(t, t)\}$ and $\mathcal{CEGBOR}_f$(N4) $= \{(t, f), (f, t)\}$. Finally, we generate the CEG-BOR test constraints given in table 3. Note that while this test constraint set has one more element than the one shown in table 2 for the TCG algorithm, the table 3 set is guaranteed to detect all Boolean operator faults that can occur in $\mathcal{E}$.

In comparison with Elmendorf's [1973] algorithm and Yokoi and Ohba's [1992] algorithm for CEG test generation, CEG-BOR algorithm has two major advantages. One advantage is that the size of a CEG-BOR test constraint set for an effect node is linear with the number of nodes in the CEG associated with the effect.

Table 3
CEG-BOR constraint set for CEG represented as $ab + c$.

| CEG-BOR test case $u$ | $\mathcal{E}(u)$ |
|---|---|
| $(t, t, f)$ | $t$ |
| $(f, t, t)$ | $t$ |
| $(f, t, f)$ | $f$ |
| $(t, f, f)$ | $f$ |

Assume that $\mathcal{CEGBOR}(\mathcal{N}_1)$ and $\mathcal{CEGBOR}(\mathcal{N}_2)$ are CEG-BOR test constraint sets for nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively. Let $\mathcal{CEGBOR}(\mathcal{N})$ be the CEG-BOR test constraint set generated for $\mathcal{N}$. Then,

1. If $\mathcal{N}$ is a cause node, $\mathcal{CEGBOR}(\mathcal{N})$ is given by $\{(t), (f)\}$.
2. If $\mathcal{N} = \mathcal{N}_1 \cdot \mathcal{N}_2$, $\mathcal{CEGBOR}(\mathcal{N})$ is constructed as follows:
$$\begin{aligned} \mathcal{CEGBOR}_t(\mathcal{N}) &= \mathcal{CEGBOR}_t(\mathcal{N}_1) \otimes \mathcal{CEGBOR}_t(\mathcal{N}_2), \\ \mathcal{CEGBOR}_f(\mathcal{N}) &= \big(\mathcal{CEGBOR}_f(\mathcal{N}_1) \times \{t_{\mathcal{N}_2}\}\big) \cup \big(\{t_{\mathcal{N}_1}\} \times \mathcal{CEGBOR}_f(\mathcal{N}_2)\big), \end{aligned} \tag{1}$$
where $t_{\mathcal{N}_1} \in \mathcal{CEGBOR}_t(\mathcal{N}_1)$, $t_{\mathcal{N}_2} \in \mathcal{CEGBOR}_t(\mathcal{N}_2)$, and $(t_{\mathcal{N}_1}, t_{\mathcal{N}_2}) \in \mathcal{CEGBOR}_t(\mathcal{N})$.
3. If $\mathcal{N} = \mathcal{N}_1 + \mathcal{N}_2$, $\mathcal{CEGBOR}(\mathcal{N})$ is constructed as follows:
$$\begin{aligned} \mathcal{CEGBOR}_f(\mathcal{N}) &= \mathcal{CEGBOR}_f(\mathcal{N}_1) \otimes \mathcal{CEGBOR}_f(\mathcal{N}_2), \\ \mathcal{CEGBOR}_t(\mathcal{N}) &= \big(\mathcal{CEGBOR}_t(\mathcal{N}_1) \times \{f_{\mathcal{N}_2}\}\big) \cup \big(\{f_{\mathcal{N}_1}\} \times \mathcal{CEGBOR}_t(\mathcal{N}_2)\big), \end{aligned} \tag{2}$$
where$f_{\mathcal{N}_1} \in \mathcal{CEGBOR}_f(\mathcal{N}_1)$, $f_{\mathcal{N}_2} \in \mathcal{CEGBOR}_f(\mathcal{N}_2)$, and $(f_{\mathcal{N}_1}, f_{\mathcal{N}_2}) \in \mathcal{CEGBOR}_f(\mathcal{N})$.
4. If $\mathcal{N} = \overline{\mathcal{N}_1}$, $\mathcal{CEGBOR}(\mathcal{N})$ is constructed as follows:
$$\mathcal{CEGBOR}_f(\mathcal{N}) = \mathcal{CEGBOR}_t(\mathcal{N}_1), \qquad \mathcal{CEGBOR}_t(\mathcal{N}) = \mathcal{CEGBOR}_f(\mathcal{N}_1). \tag{3}$$

Figure 3. Rules for CEG-BOR test generation.

The other advantage is that if the CEG associated with the effect node corresponds to singular Boolean expression, then a CEG-BOR test constraint set for the effect node guarantees the detection of incorrect “$\cdot$”/“$+$” nodes and missing “ $\overline{\phantom{x}}$ ” nodes. Tai *et al.* [1994] reported an empirical study comparing the Elmendorf, Yokoi and Ohba, and BOR algorithms. For a set of 51 non-equivalent singular Boolean expressions with four Boolean variables, the BOR algorithm is almost as effective as Elmendorf’s algorithm and is slightly better than Yokoi and Ohba’s algorithm. Also, the BOR algorithm requires about the same number of tests as Yokoi and Ohba’s algorithm and about half the number of tests as Elmendorf’s algorithm.

## 4. Application of the CEG-BOR strategy

In this section we describe some results of empirical evaluation of the CEG-BOR testing. In section 4.1 we discuss results of applying CEG-BOR to a real-time boiler control system and compare it to a strategy that generates test-cases based on finite-state machine representations. This section also illustrates how to build hierarchical CEG. In section 4.2 we compare effectiveness of the CEG-BOR strategy with that of random and functional testing. We also applied CEG-BOR to components of a commercial database system. We discuss these results in section 4.3. In all cases, we first constructed the CEG for the informal specifications, applied the CEG-BOR strategy to generate the test constraints, and then selected test cases that satisfied these test constraints.

## *4.1. A safety-related real-time system*

The specification of a simplified real-time boiler control and monitoring system was 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 [Institute for Risk Research 1993]. One version of the boiler control and monitoring system was developed at North Carolina State University [Paradkar *et al.* 1993; Vouk and Paradkar 1993]. During the development of the boiler system, the original informal specification of the system [Institute for Risk Research 1993] was re-written in terms of several extended finite-state machines[1] (EFSMs). These EFSMs did not represent concurrent processes, but represented sequential processes. Test suites for the unit, integration, and system testing of the boiler system were constructed according to the boiler's EFSM specification [Paradkar *et al.* 1993]

The purpose of this experiment was to compare the CEG-BOR test set and the EFSM-based test set. We limited our experiments only to the most critical boiler system behavior, the *boiler shutdown* effect. Below we briefly describe the boiler system.

The *simplified* boiler system used in the study consists of a natural gas fired watertube 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 a lower limit. If the water level is above the upper limit, water will be carried over into the steam flow and it will 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. The readings from the 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 the potential errors, and issue a correction/repair request if any errors are discovered.

[1] An extended FSM is an FSM with the use of variables.

3. To display, for the boiler operator, the “best estimates” of the different parameters.

4. To accept any appropriate operator commands.

#### *4.1.1. 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 using the hierarchical approach. The number of levels is governed by the complexity of the CEG at each level. Thorough understanding of the specification is very important in making this choice. Wrong choice of intermediate nodes could exacerbate the complexity instead of mitigating it. In this particular case, the choice of levels was natural. In general, this may not be the situation. We discuss these issues using the shutdown CEG as the example. The level 1 (the topmost/highest level) specification-CEG for boiler shutdown is shown in figure 4. Brief descriptions of the nodes at this level are shown in figure 5.

The cause nodes of the level 1 specification-CEG, including C180, C181, C190, and C192 through C196, are effect nodes of level 2 CEG. For example, C193 which represents communication link failure, is an effect of several causes. These include no transmission in a 5 second interval, illegal command, illegal data format etc. This is a natural choice for level 2 CEG corresponding to C192. The illegal data format, in turn, can be decomposed further. For the level 3 CEG corresponding to illegal data format node, the input causes are data not left justified, data out of range, data with multiple decimal points, etc. Similarly, some of the other cause nodes of level 2 CEG are effect nodes of level 3 CEG and so on. It should be noted, that not all the nodes at a given level are decomposed further. For example, C198 is a terminal node in level 1 CEG and it is not decomposed any further.

As mentioned earlier, the boiler problem was re-specified using EFSMs. This represented the process view of the boiler system, where each EFSM corresponded to a single component of the system. However, we found this representation inadequate when we wanted to visualize the interactions of the processes that lead to a shutdown. Instead, we used CEG. Our experience is that the CEG approach is more flexible and practical because it allows the designers to focus on the most critical aspect of the system. This unified view of the system assists in detecting ambiguities and inconsistencies.

#### *4.1.2. Comparison of CEG-BOR- and EFSM-based test sets*

The original test set was constructed using an extended finite state machine (EFSM) representation. However, due to the project duration constraints, exhaustive testing of all the state transitions in these EFSMs was not feasible. To manage the complexity, the original developers made some assumptions. For example, they assumed that more than two faults which mask each other do not occur simultaneously. Since we only constructed the CEG for the shutdown effect, we selected the subset of the EFSM-based test set for the shutdown effect. We studied the following two issues:

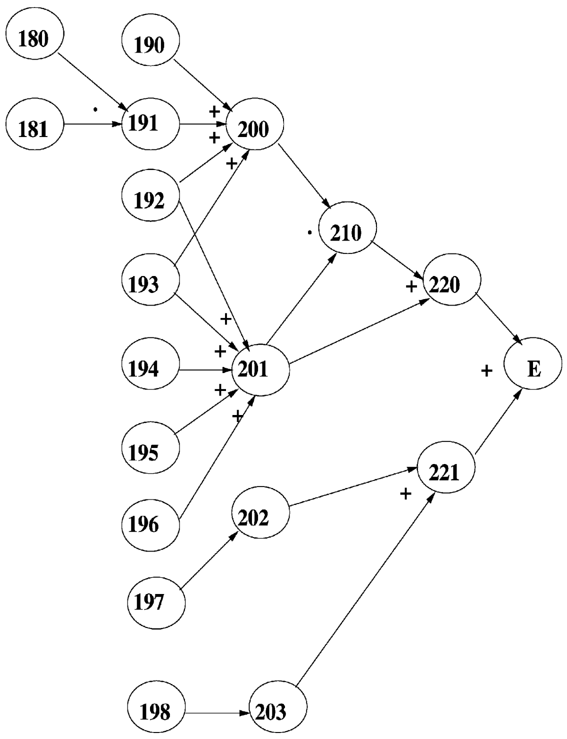

Figure 4. Level 1 specification cause-effect graph for boiler control and monitoring system.

E – boiler shutdown
C221 – externally initiated
C220 – internally initiated
C203 – instrumentation system initiated
C202 – operator initiated
C201 – bad startup
C200 – operational failure
C198 – confirmed “shutnow” message
C197 – confirmed keystroke entry
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
C181 – steam rate meter failure during operation
C180 – water level meter failure during operation

Figure 5. Brief descriptions of the level 1 nodes for the boiler control system.

- Does the EFSM-based shutdown test set satisfy the BOR criterion for the shutdown CEG?
- Is the CEG-BOR test set more effective than EFSM-based shutdown test set for fault detection?

The selected EFSM-based test set, referred to as the shutdown test set, contained 372 test cases. We then measured if this EFSM-based test set satisfied the BOR criterion for the specification-CEG. Of the 372 tests in the shutdown test set, 59 tests (about 1/6 of the total) were found to be redundant, in that, more than one test satisfied the same BOR constraint. But 24 more test cases were needed to satisfy BOR criterion completely. This implies that out of the 337 ($= 372 - 59 + 24$) test constraints required to satisfy the BOR criterion only 313 ($= 372 - 59$) were satisfied by the EFSM test set. The EFSM shutdown test set was constructed by three persons who spent a total of approximately 100 person-hours while the shutdown specification-CEG was constructed by one person in about 20 hours. Furthermore, the CEG-based test-case generation can be automated [Paradkar *et al.* 1996].

It is worth noting that the additional tests needed for 100% satisfaction of the BOR criterion for the shutdown CEG detected a fault in the boiler's implementation that was not detected by the EFSM-based test set. The implementation module which contained the fault has 360 C language statements and 34 conditional statements. Out of these 34 conditional statements, 21 are atomic (that is, there are no "$\cdot/+$" operators) and the remainder are expressions with one "$\cdot/+$" operator. 81 test constraints would be needed to satisfy BOR criterion for testing these conditional statements (two test constraints are needed for each atomic expression and three test constraints for each expression with one "$\cdot/+$" operator). Two of the 81 constraints were not covered by the EFSM-based shutdown test set. The undiscovered fault in the module was associated with one of these two constraints. The BOR criterion was not satisfied because of the assumptions that were made during the construction of the test set. On the other hand, the CEG-BOR generated test cases did satisfy the BOR criterion and were able to exercise the fault.

### 4.2. *Functionally equivalent software*

Six functionally equivalent Pascal programs were developed as part of a NASA-supported study on software testing and fault-tolerance [Vouk *et al.* 1986a][2]. They solved a simplified satellite navigation problem – an extended version of the "Earth satellite problem" used by Nagle and Skrivan [1982]. Size of these Pascal programs ranged from 400 to 800 statements. Acceptance testing of the developed software involved both random and special functional test cases. All acceptance test data were generated on the basis of functional problem specifications. Major and minor func-

[2] N-version programming is a technique for increasing software reliability. The basic idea 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.

tionalities of the system were tabulated, as were all system input, output, and primary intermediate variables (including legal range of variables). Test cases were generated using these tables. Special consideration was given to both explicit and implicit extremal and special values in the specification and in the problem solution algorithms (boundaries, singularities, etc.). Random data were generated using a uniform distribution for all input parameters. The nature of failures and faults detected in the components during the acceptance testing phase are described in [Vouk *et al.* 1986a, b]. For technical reasons, only five of those program were used in the current experiment.

The objectives of this experiment were to:

- compare fault detection properties of the CEG-BOR, random, and functional test sets for these programs;
- evaluate relative cost-effectiveness of these strategies (i.e., the number of faults detected per number of test-cases generated by the strategy);
- evaluate the robustness of these testing strategies across functionally equivalent software. This also gave us an insight into the the potential relationship between the fault detection properties of a methodology and the structural coverage it generates.

In this study the coverage measurements were automated using a tool called BGG. This tool can measure the test coverage of statements, branches, and various types of data flow metrics for Pascal programs. It was developed at NCSU [Vouk and Coyle 1989].

We first give a brief description of the navigation problem. Then, we present the code coverage measurements, followed by a comparison of the fault detection properties of the strategies we used. Finally, we discuss the implications our results have on the relationship between fault detection properties of a testing strategy and code coverage.

### *4.2.1. Satellite navigation problem*

The gist of the problem [Nagle and Skrivan 1982] is that given two points on the surface of earth, one needs to compute the distance between them and generate the shortest path between these points in terms of intermediate coordinates. Part of this specification is reproduced here:

> *"... on exit PATHPOINT* [1], *..., PATHPOINT* [*NPOINTS*] *represent point along a great circle from the first point to the second. The sequence represents coordinates for N (adjusted if necessary) equally-spaced points along the great circle path from the first point to the second, beginning with the first, except that points on or inside the cone determined by the third point and an angle a should be excluded. Note that if P1 and P2 are coincident, as many as N identical points may be returned. If the required sequence of points cannot be determined uniquely from the input parameters, NPOINTS should be set to* $-1$ *upon exit, and PATHPOINT assumed to*

*have no particular meaning. This could occur if, for example, the two points were the North Pole and the South Pole and N had the value* 4.*"*

The construction of the CEG for this specification was not difficult. However, the numerical aspects of the specification were not explicitly modeled by the CEG. The evaluation of the numerical precision requirements was done explicitly through selection of appropriate test cases that satisfied both the CEG-BOR criterion and the numerical validation criteria.

As in the boiler study, construction of the CEG revealed an ambiguity in the specification. In particular, the last line of the specification excerpt above does not say what the action should be when two points are specified (North and South Poles) and N has the value of 2. It is obvious that this situation does not fit the description of the exception. It is possible to return the two endpoints in the required PATHPOINT variable. Upon consultation with the experimenters, it was found that at the time of the experiment, the ambiguity was resolved as per the interpretation above (valid values should be returned). However, some of the programmers did not implement this solution, and the original test-suite did not contain test-cases to detect this fault. This triggered the fault in some of the programs when the CEG-BOR based tests were run.

### *4.2.2. Coverage*

Satisfaction of the CEG-BOR criterion for the satellite problem required the generation of 43 test cases. Some of the test cases dealing with precision in the software could not be derived from the CEG alone. But, since answer precision was implicit in the specification, the constraints that satisfied precision requirements were generated explicitly.

The "traditional" code coverage provided by the 43 CEG-BOR test cases was practically identical to that provided by the functional test suite of 103 test cases generated using black-box techniques. Table 4 summarizes the average coverage measures over these five programs. Since CEG-BOR testing required about half as many test cases as the "traditional" functional test-suite, the specification-based CEG-BOR test selection could have considerable advantages in terms of costs, provided that code coverage was an accurate indicator of the value of the methodology in terms of its fault detection properties. We discuss this in the next subsection.

Table 4
Average coverage of various metrics by different methods.

| Method | Statement coverage | Branch coverage | p-use coverage |
|---|---|---|---|
| Random | 0.85 | 0.63 | 0.61 |
| Functional | 0.96 | 0.90 | 0.87 |
| CEG-BOR | 0.96 | 0.90 | 0.88 |

One goal of this study was to investigate the behavior of the three specification-based testing strategies under implementation diversity. Each of the five programs was tested (using individual drivers) with three test-suites: one containing 1,000 randomly selected test cases, the second containing 103 manually designed functional test cases (i.e., special value functional tests), and the third containing the 43 test cases developed using the CEG-BOR strategy. For each program we measured the coverage of p-uses (a data flow testing metric reported in [Frankl and Weyuker 1988; Weyuker 1988]), branches, and statements. The branch and p-use coverage dynamics are illustrated in figures 6–9. The programs are marked L1 through L5. In all figures, we see the usual metric saturation characteristic of a single strategy testing approach. But, we also see one other interesting phenomenon – the variability in the coverage due to the implementation diversity. For example, all three strategies provided branch coverage that differed by as much as 20% between some versions. This variability in the code coverage furnishes further evidence that specification-based and code-based testing are complementary. Code-coverage should be measured for specification-based test cases and additional code-based test cases should be developed to attain the desired code-coverage values.

### *4.2.3. Fault detection capabilities*

The fault-detection power of the CEG-BOR strategy needs to be comparable to that of the other strategies for the CEG-BOR strategy to be useful in practice. Functional testing had discovered a total of five distinct faults in all five programs. Random testing had discovered a total of three distinct faults in these five programs (these faults were different from the ones found in functional testing). The five programs had, thus, a total of eight distinct known faults. It is interesting to know that the fault sets detected by functional and random testing are disjoint. In our experiments the CEG-BOR data set detected all the known faults in all five programs. In addition to detecting the known faults, the CEG-BOR test set also discovered an additional fault in two programs. This fault could be attributed to the ambiguity in the specification described earlier. It is obvious, at least in this case, that CEG-BOR fault detection capabilities are superior to that of the other two strategies. What is interesting to note, though, is that both the CEG-BOR and the functional testing provided similar code coverage while random testing provided code coverage that was about 10 to 25% lower. This would seem to suggest that the relationship between the code coverage and the fault detection is not so obvious. In general, while, very low coverage of more complex metrics (such as p-uses) may be an indicator that a test-generation technique may not have good fault detection properties, there is no evidence that pursuit of high-coverage will improve fault detection, particularly when very simplistic metrics such as statement coverage are used.

The experiment-specific fault-detection dynamics are illustrated in figure 10. It shows the plot of the total number of faults detected by the CEG-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 CEG-BOR testing detected all faults that

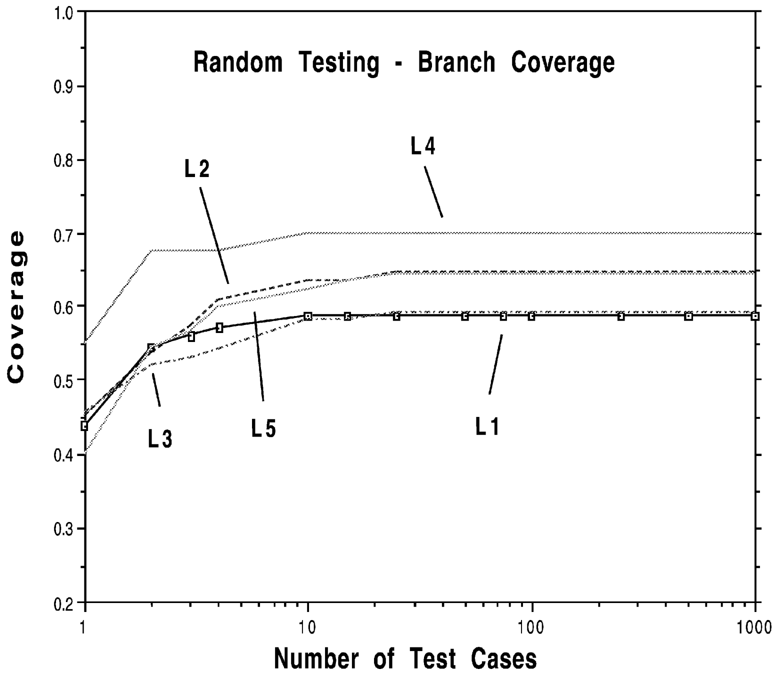

Figure 6. Branch coverage of 5-version programs provided by random testing.

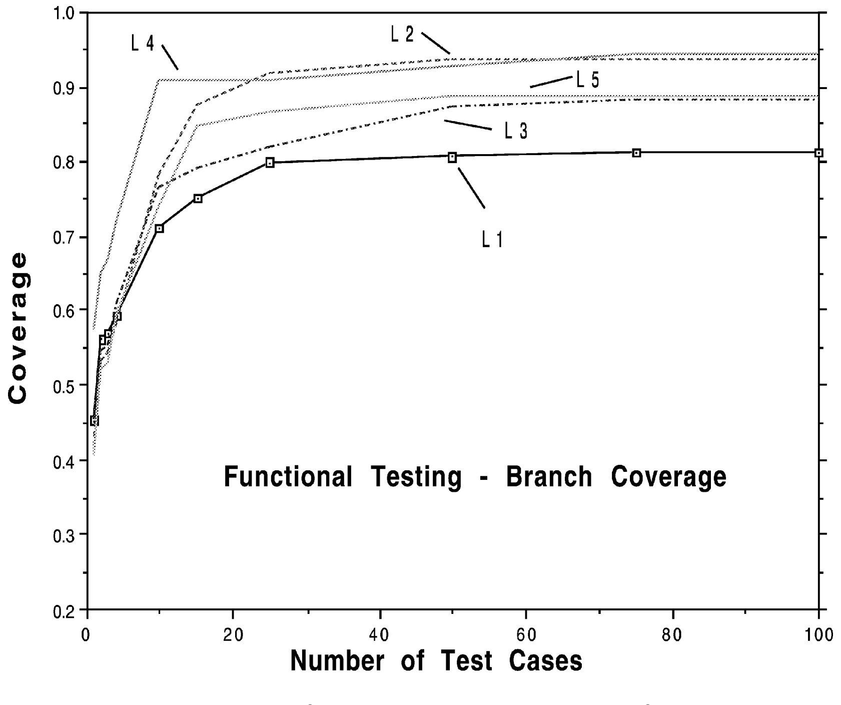

Figure 7. Branch coverage of 5-version programs provided by functional testing.

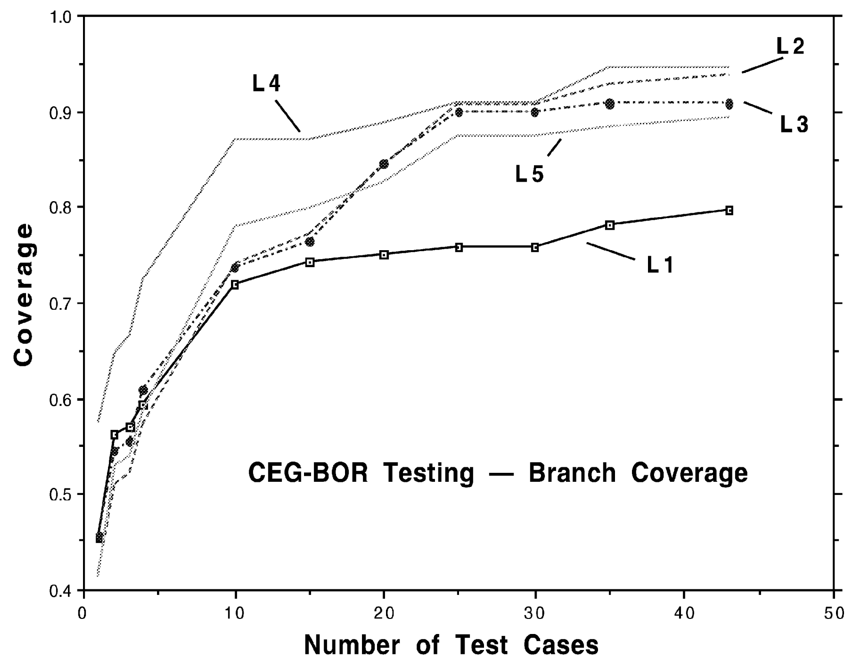

Figure 8. Branch coverage of 5-version programs provided by CEG-BOR testing.

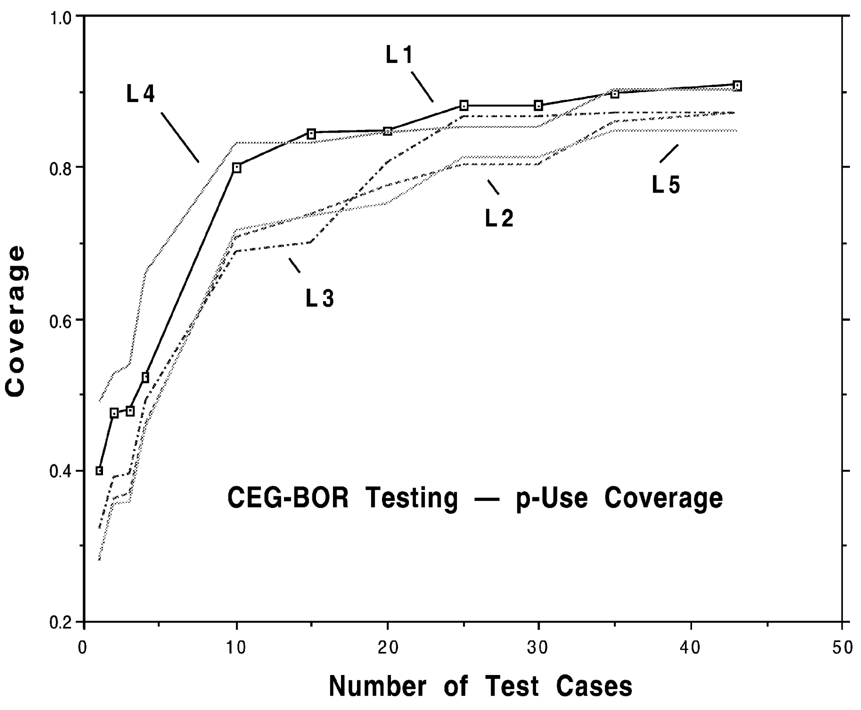

Figure 9. p-use coverage of 5-version programs provided by CEG-BOR testing.

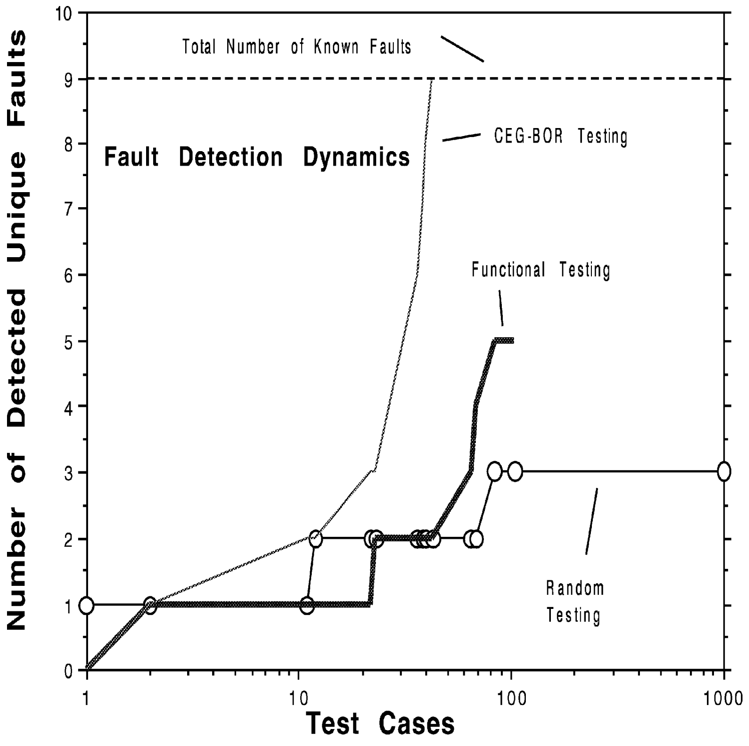

Figure 10. Fault detection in 5-version programs by CEG-BOR, random, and functional testing.

functional and random testing did together. Furthermore, it did that with a smaller number of test cases than either functional or random testing.

### 4.3. *Commercial data base software*

In addition to assessing CEG-BOR in an academic environment, we also evaluated the CEG-BOR strategy in an industrial environment. Of particular concern was the issue of applicability of CEG-BOR strategy to real projects. We used the CEG-BOR strategy on two components of a commercial database system. We discuss the results in the following sections[3].

#### 4.3.1. *Database monitor*

The first application we analyzed was a system used to collect statistics on the operation of the management component of a large commercial database system. The application was several tens of thousands of lines of high level code (in C) in size. For the purpose of this experiment, we wanted to formalize the specifications at the level of requirements. Since the functional specification was available only as a detailed design

[3] To preserve confidentiality, this discussion does not name the products and the organization that built them.

document, we reconstructed this high level abstraction through functional composition of the detailed design. Based on the analysis we constructed a CEG representation of this specification. We identified four distinct groups of CEG corresponding to four main functions of the product. In general, functional reconstruction from a design document is considered a good way of verifying the design against the original specifications as well as a means of checking it for ambiguity. This proved to be true here as well. Our CEG construction uncovered ambiguities in the design. These ambiguities did not, however, propagate into the implementation because of the extensive testing that followed in practice. We compared the original product test suite with the test suite designed and implemented based on this higher level CEG representation of the product. We found that 89 of the original test cases could have been removed and that we would still have satisfied the CEG-BOR criterion and would have uncovered all the faults the original test suite detected.

#### *4.3.2. Concatenation operator*

The second element of the database system that we examined was a 5000+ lines of high level code that implemented the SQL concatenation operation. The CEG for this application was constructed based on the original functional specification. During the construction of the CEG, we uncovered an ambiguity in the specification. Detailed analysis of the faults discovered by the test cases originally generated for this product showed that this ambiguity did not propagate to the implementation shipped to the customer, again because of the extensive implementation level testing. Our CEG based test suite was only about half as big (104 test cases) as the original test suite (205 test cases) but could uncover the same functional faults as the original test suite.

These two case studies have increased our belief in the applicability and fault detection capabilities of CEG-BOR methodology. It appears that in both cases CEG-BOR strategy would have saved considerable effort (judging by the number of test cases) without any loss in fault detectability. In addition, the process of CEG construction also appears to be useful in uncovering ambiguities and inconsistencies in the specifications in early software development stages.

## 5. Limitations of CEG-BOR

Despite the encouraging results reported in the previous section, the CEG-BOR strategy has several limitations. One of the limitations of the CEG-BOR strategy has to do with whether or not a Boolean expression is singular. In the CEG-BOR strategy, each cause is treated as an independent Boolean variable. This may lead to generation of infeasible test constraints. For example, consider the specification "Return error if $X$ is greater than 10 and $Y$ is less than 5 or if $X$ is not greater than 10 and $Z$ is greater than 15." CEG-BOR would convert this into a Boolean expression $ab + \bar{c}d$ with Boolean variables $a$ and $c$ representing the same atomic sentence "$X$ is greater than 10". We may generate an infeasible test constraint $(t, f, f, t)$ (since it contains conflicting values for equivalent atomic sentences). In the empirical studies reported in section 4 we did not encounter such infeasible constraints, but this may not be true in general.

To deal with this limitation, we first must modify the definition of a test constraint for a Boolean expression. Instead of associating a symbol with each literal in the expression, we associate the symbol with each distinct Boolean variable in the expression. Next, we combine the BOR strategy with another strategy called the Meaningful Impact strategy. The Meaningful Impact (MI) testing strategy for Boolean expressions was reported in [Weyuker *et al.* 1994]. It can be applied to singular or non-singular Boolean expressions. The strategy is based on the detection of missing and/or extra negation operators on individual variables. Weyuker *et al.* [1994] reported results of an empirical study done on twenty specifications written as Boolean expressions. The authors reported very good fault detection rates for different types of faults. Still, even though the strategy focuses on the detection of missing/extra negation operators, it cannot guarantee detection of all such faults in an expression because it uses the sum-of-product representation of the expression, where a Boolean expression is represented by *sums* (which are subexpressions joined by "$+$" operators) of *product terms* (which are literals joined by "$\cdot$" operators), to develop the test cases. Also, for singular expressions, the MI strategy generates a larger test constraint set than does the BOR strategy.

In [Paradkar and Tai 1995], we presented a new algorithm, called BOR+MI, that combines the BOR and Meaningful Impact (MI) [Weyuker *et al.* 1994] strategies. This hybrid algorithm partitions an input Boolean expression into components such that we can apply the BOR strategy to some of these components and the MI strategy to the remaining components. The test constraints for individual components are combined using the BOR strategy. Analytical and empirical results reported in [Paradkar 1996; Paradkar and Tai 1995] indicate that:

- the BOR+MI algorithm usually produces a smaller test constraint set for a Boolean expression than does the MI strategy,
- the BOR+MI strategy and the MI strategy have comparable fault detection capability.

Another limitation of the CEG-BOR strategy is that it generates only the test constraints and not the test cases. The CEG-BOR strategy can also be extended for automatic generation of actual test cases. To accomplish this goal we refined the granularity of representing the atomic sentences in a specification (with the restriction that all the input variables are numeric). This refinement allows us to capture numerical relationships like "$X$ is greater than 5" in an atomic sentence using expressions like $X > 5$. Such a representation can be manipulated to first generate the test constraints and then the test cases using constraint logic programming techniques. We have reported such an extension to CEG-BOR in [Paradkar *et al.* 1996].

## 6. Summary and conclusions

In this paper we have presented the CEG-BOR algorithm for specification based testing. Of special interest were three properties of the methodology: its cost, its ability to detect faults, and its range and robustness (that is, its ability to consistently provide this fault detection capability across different software application areas and under different conditions).

It is our experience that hierarchical CEG can provide an efficient way of controlling the complexity and the cost of construction of cause-effect graphs that can arise in real applications. The choice of intermediate nodes may not be obvious at all times, but we believe that an experienced analyst can judge effectiveness of various options fairly easily. Knowledge of the application domain is a very critical factor during the construction of the cause-effect graphs. To uncover faults, ambiguities, and inconsistencies in the detailed design, it may be beneficial to augment the requirements-based CEG with functional and structural information generated during the detailed design phase.

In our experience, the CEG-BOR approach is at least as effective in detecting faults as either informal state-based testing or thorough but informal functional and random testing. This behavior was observed in three different application areas and across functionally equivalent programs. One reason is that the process of CEG specification analysis highlights ambiguities and incompleteness in the specifications and forces examination of boundary and special value conditions. In turn this helps isolate input space failure sets [Musa *et al.* 1987] in a way that may not be possible with purely structural and program-based testing methods. In practice, the CEG-BOR testing may need to be augmented by other techniques to improve its fault detection property for non-singular expressions.

The code coverage provided by the CEG-BOR testing (e.g., statement, branch, or p-use coverage) was in all our case studies higher than the coverage provided by the other techniques we examined. However, none of the specification-based strategies we investigated provided 100% code coverage of the implementation code. Furthermore, we have noticed that the relationship between the coverage provided by a technique, and its fault detection capabilities was not well expressed. In fact, code coverage provided by exactly the same test-suite across functionally equivalent software units differed by as much as 10–20%. This suggests that extra caution should be exercised when using code coverage based test stopping criteria. The value of a testing methodology should not be judged on the basis of its code coverage properties unless the code coverage actually carries some fault detection guarantees.

## Acknowledgements

We would like to thank the referees for their constructive comments and suggestions.

## References

Basili, V.R. and R.W. Selby (1987), "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering SE-13*, 12, 1278–1296.

Beizer, B. (1990), *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, New York, NY.

Bender, R.A. (1991), *Requirements Based Testing CASE Tool*, SoftTest, Bender and Associates, Larkspur, CA.

Boehm, B. (1989), *Tutorial: Software Risk Management*, IEEE Computer Society Press, Washington, DC.

Elmendorf, W.R. (1973), "Cause-Effect Graphs in Functional Testing," TR-00.2487, IBM Systems Development Division, Poughkeepsie, NY.

Foreman, L. and S.H. Zweben (1993), "A Study of the Effectiveness of Control and Data Flow Testing Strategies," *Journal of Systems and Software 21*, 3, 215–228.

Frankl, P. and E.J. Weyuker (1988), "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering 14*, 10, 1483–1498.

Hamlet, R. (1995), "Software Testing," In *Advances in Computers*, M.V. Zelkovitz, Ed., Academic Press, San Diego, CA, pp. 1402–1411.

Howden, W.E. (1987), *Functional Program Testing and Analysis*, McGraw-Hill, New York, NY.

IEEE-ANSI (1989), *IEEE Guide for the User of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Std 982.2-1988, IEEE Standards Board and ANSI, New York, NY.

Institute for Risk Research (1993), *Generic Problem Competition: A Component of the International Symposium on Design and Review of Software Controlled Safety-Related Systems*, Software Specification for the Generic Problem Competition, Ottawa, Ontario, Canada.

Konz, S. (1995), *Work Design – Industrial Ergonomics*, Fourth Edition, Publishing Horizons, Inc., Scottsdale, AZ.

Laski, J. (1989), "Testing in the Program Development Cycle," *Software Engineering Journal 4*, 2, 95–106.

Lyu, M. (1994), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill, New York, NY.

Musa, J.D., A. Iannino and K. Okumoto (1987), *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, NY.

Myers, G.J. (1979), *The Art of Software Testing*, Wiley, New York, NY.

Nagle, P.M. and J.A. Skrivan (1982), "Software Reliability: Repetitive Run Experimentation and Modeling," Bsc-40336, Boeing Inc., Seattle, WA.

Nursimulu, K. and R.L. Probert (1995), "Cause-Effect Graphing Analysis and Validation of Requirements," In *Proceedings of CASCON'95*, IBM Canada Ltd. and National Research Council, Toronto, Ontario, Canada, p. 293.

Paradkar, A.M. (1996), "Specification Based Testing Using Cause-Effect Graphs," PhD Dissertation, Department of Computer Science, North Carolina State University, Raleigh, NC.

Paradkar, A.M., I. Shields and J. Waters (1993), "The NCSU Solution to the Generic Problem Exercise: Boiler Control and Monitoring System," Software Documentation for NCSU Solution to the Generic Problem Exercise, Department of Computer Science, North Carolina State University, Raleigh, NC.

Paradkar, A.M. and K.C. Tai (1995), "Test Generation for Boolean Expressions," In *Proceedings of International Symposium on Software Reliability Engineering'95*, IEEE Computer Society Press, Los Alamitos, CA, pp. 106–115.

Paradkar, A.M., K.C. Tai and M.A. Vouk (1996), "Automatic Test Generation for Predicates," *IEEE Transactions on Reliability 45*, 4, 515–530.

Tai, K.C. (1996), "Theory of Fault-Based Predicate Testing for Computer Programs," *IEEE Transactions on Software Engineering 22*, 8, 552–562.

Tai, K.C., M.A. Vouk, A. Paradkar and P. Lu (1994), "Evaluation of a Predicate-Based Software Testing Strategy," *IBM Systems Journal 33*, 3, 445–457.

Vouk, M.A. and R.E. Coyle (1989), "BGG: A Testing Coverage Tool," In *Proceedings of 7th Annual Pacific Northwest Software Quality Conference,* Lawrence and Craig, Inc., Portland, OR, pp. 212–233.

Vouk, M.A., M.L. Helsabeck, D.F. McAllister and K.C. Tai (1986a), "On Testing of Functionally Equivalent Components of Fault-Tolerant Software," In *Proceedings of Computer Software and Applications Conference (COMPSAC'86)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 414–419.

Vouk, M.A., D.F. McAllister and K.C. Tai (1986b), "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software," In *Proceedings of Workshop on Software Testing*, IEEE Computer Society Press, Los Alamitos, CA, pp. 74–81.

Vouk, M.A. and A.M. Paradkar (1993), "Design and Review of Software Controlled Safety-Related Systems: The NCSU Experience with the Generic Problem Exercise," In *Proceedings of the 1993 International Invitational Workshop on Design and Review of Software Controlled Safety-Related Systems*, Institute for Risk Research, Ottawa, Ontario, Canada, pp. 209–222.

Weyuker, E.J. (1988), "An Empirical Study of the Complexity of Data Flow Testing," In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, IEEE Computer Society Press, Los Alamitos, CA, pp. 188–195.

Weyuker, E.J., T. Goradia and A. Singh (1994), "Automatically Generating Test Data from a Boolean Specification," *IEEE Transactions on Software Engineering 20,* 5, 353–363.

White, L. (1995), "Software Testing and Verification," In *Encyclopedia of Microcomputers*, Volume 16, A. Kent and J. Williams, Eds., Marcel Dekker Inc., New York, NY, pp. 185–241.

Yokoi, S. and M. Ohba (1992), "TCG: CEG-Based Tool and Its Experiments," In *Proceedings of the 13th Software Reliability Symposium*, IEEE Computer Society Press, Los Alamitos, CA, pp. 41–49.