Abstract. BZ-TESTING-TOOLS (BZ-TT) is a tool-set for automated test case generation from B and Z specifications. BZ-TT uses boundary and cause-effect testing on the basis of the formal model. It has been used and validated on several industrial applications in the domain of critical software, particularly smart card and transport systems. This paper presents the test coverage criteria supported by BZ-TT. On the one hand, these correspond to various classical structural coverage criteria, but specialised to the case of B abstract machines. The paper gives algorithms for these in Prolog. On the other hand, BZ-TT introduces new coverage criteria for complex data structures, based on boundary analysis: this paper defines weak and strong state-boundary coverage, input-boundary coverage and output-boundary coverage. Finally, the paper describes how BZ-TT presents a unified view of these criteria to the validation engineer, and allows him or her to control the test case explosion on a coarse basis (choosing from a range of coverage criteria) as well as a fine basis (selecting options for each state or input variable).

Keywords: model-based testing, boundary values, set constraint solving, B notation.

1 Introduction

Test adequacy criteria [1] play an increasingly important role in code-based software testing practice. A wide range of criteria, based mainly on control flow or data flow, helps to select test suites or to measure their quality.

In contrast, specification-based (or black-box) testing is more typically guided by testing strategies, such as category-partitioning, syntax-testing, cause-effect or boundary testing [2]. Currently, the industrial practice for black-box testing is for a validation engineer to manually design test cases on the basis of the technical requirements documentation. The drawbacks of this approach are well-known: there are no clear rules that determine when sufficient testing has been done. The quality of black-box testing depends essentially on the know-how of the validation engineer, with poor rationale and reproducibility.
The need to offer better methods and tools for specification-based testing has given rise to a large amount of research on generating tests from formal specifications. Formal methods of specification, and particularly model-oriented notations such as Z and B [3], allow a high-level abstract formalization of the expected behavior of the system under test. These notations are well-suited for test generation because the expressiveness of set-oriented logic constructs and the definition of an explicit model help both test case generation and oracle synthesis. Thus, these formal notations are the basis of various proposals to more or less automatically generate tests from the formal model, see for example [4–8].

A new method for automated test generation from B abstract machines or Z specifications was presented in [9]. This method, called BZ-TESTING-TOOLS, uses cause-effect analysis and boundary computation to produce test cases as sequences of operation invocations. This computation is based on customized constraint logic programming techniques [10] both for extracting boundary values and for sequencing operation invocations. This test generation method is embedded in the BZ-TESTING-TOOLS environment [11], which has been exercised on several industrial applications in the domain of smart card software (a GSM 11-11 application [12, 13], the Java Card transaction mechanism [14]) and for transport applications (a Metro/RER ticket validation algorithm and an automobile windscreen-wiper controller). In all these applications, a B abstract machine was built specifically for automatic test generation, by an independent validation team. There was no formal specification for the whole system, just informal requirements, for example the GSM 11-11 standard [15]. Writing a specific specification for testing has been shown to be cost-effective [13], and has the advantage that it can be tailored towards the desired test objectives. The formal model is proved using Atelier B [16] and validated using constraint animation with the BZ-TT tool-set before test case generation. The connection with specific test beds is done by translating abstract generated test cases and oracles into executable test scripts [14], enabling automated test execution and verdict assignment.

This article focuses on control-flow and data-oriented coverage criteria for B abstract machines and how one can use it to control the test generation process in the BZ-TT environment. Indeed, reducing and controlling the test case explosion problem is a key issue for model-based test generation. The proposal of the BZ-TT method and tools for dealing with this dreaded problem is on the one hand to allow a systematic minimal test generation achieving strong coverage results, but reducing the number of tests as much as possible, preferably to a linear number. On the other hand, the idea is to allow the test engineer to focus on specific areas of the specification, using a hierarchy of options to expand the test coverage of that area, using well-defined and understandable coverage criteria, while still controlling test case explosion.

Section 2 introduces a structural analysis framework for B abstract machines. Section 3 applies the classical control-flow graph criteria from imperative programs to these abstract models, obtaining a family of cause-effect coverage criteria. Section 4 introduces a family of boundary-oriented coverage criteria which choose tests from the boundaries of a given state space. Section 5 gives an overview of the BZ-TT environment and the default test generation process, then Section 6 describes the hierarchy of options that allow a test engineer to control the test case explosion, measure coverage, and focus
attention on specific areas. Section 7 demonstrates the approach on the classic triangle example [2] and describes results from large industry case studies. Section 8 describes related work and Section 9 presents conclusions and future work.

2 Control-Flow Analysis for B Abstract Machines

Our goal is to generate tests from an abstract formal model of some implementation that is developed independently. The formal model is typically written for the purposes of testing, to satisfy specific test objectives, test certain points of control and observations of data.

The BZ-TT environment supports B abstract machines [3] and several other formal specification notations [17]. The BZ-TT tool-set requires some restrictions on the input specification. Firstly, it must specify a single machine. For B, this means that only one abstract machine is allowed, without layering. Secondly, operations must have explicit preconditions. In B, operations usually have explicit preconditions, but the BZ-TT approach requires the entire precondition to appear at the beginning of the operation, and also requires this precondition to be strong enough to ensure that the operation is feasible. Thirdly, all data structures must be finite, which means that the given sets are either enumerated or of a known finite cardinality. Fourthly, the B control structure must be deterministic (but note that individual effects may still be non-deterministic, if they use the ANY operator or non-deterministic assignment). This assumption makes it easier to compare the coverage results with the traditional code coverage criteria, where the control structure is also deterministic. During industrial trials, these restrictions have not been a problem.

The translation scheme from B generalised substitutions to before-after predicates is precisely defined in the B-Book [3]. As a running example, the B operation shown in Figure 1 is used, which contains a variety of B constructs ($C_i$ are atomic predicates and $Sub_j$ are elementary substitutions).

The body of each operation is translated into a before-after predicate. Basically, it consists of unfolding predicates along branches, and introducing primed variables to denote the after values, using the prd rules from [3, Chap. 6]. Each choice operator is translated into a predicate choice operator $\mathcal{L}$, which is semantically equivalent to disjunction. The reason for using a separate choice operator rather than using disjunction everywhere is that it enables the control structure of the B specification to be analysed independently of any disjunctions within conditions. This is quite different to the usual approaches based on DNF partitioning [4] and produces fewer alternatives.

The parallel substitution is translated using the following rule:

$$\text{prd}_{x,y}(S \parallel T) = \text{prd}_x(S) \land \text{prd}_y(T)$$

Note that this prd rule means that the state variables are partitioned along the two branches of each parallel operator, so that each leaf of the tree is associated with a disjoint subset of the state variables. The elementary substitution ($x := E$) at the leaves becomes the before-after predicate $x' = E$, and $\text{prd}_x(\text{skip})$ is $x' = x$. 

\[ \text{Op} \equiv \begin{align*}
\text{PRE} & \quad \text{PRE}_1 \lor \text{PRE}_2 \\
\text{THEN} & \quad \text{IF } C_{10} \land C_{11} \land C_{12} \\
& \quad \text{THEN } \text{Sub}_1 \\
& \quad \text{ELSE } \text{Sub}_2 \parallel \\
& \quad \text{SELECT } C_2 \land C_3 \land C_4 \text{ THEN } \text{Sub}_3 \\
& \quad \text{WHEN } C_5 \text{ THEN } \text{Sub}_4 \parallel \text{Sub}_5 \end{align*} \]
\[ \text{END} \parallel \text{END} \]

\[ \text{IF } C_6 \lor C_7 \text{ THEN } \text{Sub}_6 \]
\[ \text{END} \]
\[ \text{END} \]

**Fig. 1.** Example of an Operation in B

The example in Fig 1 gives the following predicate:

\[
( C_{10} \land C_{11} \land C_{12} \land \text{prd}(\text{Sub}_1) \\
\text{[} (\neg C_{10} \lor \neg C_{11} \lor \neg C_{12}) \land \text{prd}(\text{Sub}_2) \land \\
( C_2 \land C_3 \land C_4 \land \text{prd}(\text{Sub}_3) \\
\text{[} C_5 \land \text{prd}(\text{Sub}_4) \land \text{prd}(\text{Sub}_5) \\
\text{)} \land \\
( (C_6 \lor C_7) \land \text{prd}(\text{Sub}_6) \\
\text{[} \neg (C_6 \lor C_7) \land \text{prd}(\text{skip}) \\
\text{)}
\]

**Fig. 2.** Control-Flow Graph of the Before-After Predicate Resulting from Figure 1

This can be viewed as a specific kind of control-flow graph (see Fig 2) where the arcs out of a node are alternative choices, each arc contains a decision predicate conjoined with a substitution in predicate form, and each path through the graph is the conjunction of its arcs. Note that it is a very restricted form of control flow graph:
– Since B abstract machines have no loops, the control-flow graph has no loops.
– Since the conjunction operator is commutative, the parallel subgraphs can be evaluated in either order (note that conjunction represents parallelism, not sequencing – B abstract machines do not allow sequencing). For example, the C6/C7 subgraph could be traversed first rather than last.
– Our assumption that the control flow structure is deterministic means that each choice statement has mutually exclusive branches. Thus, when one branch of a B choice statement is true, the others are false.

After translating the operation to a predicate, the operators (but not the \( \lor \) operators) are propagated up to the top level, using the following distributive laws.

\[
A \land (B \parallel C) \sim A \land B \parallel A \land C
\]
\[
(A \parallel B) \land C \sim A \land C \parallel B \land C
\]

These transformations result in a postcondition \( E_1 \parallel \ldots \parallel E_n \), where each \( E_i \) is a before-after predicate that does not contain \( \parallel \) operators. This form of postcondition is called Effect Disjunctive Normal Form (EDNF), because each disjunct corresponds to one effect (or behavior) of the operation, which is one path through the control-flow graph. The set of all the EDNF predicates corresponds to the set of all control paths through the original B operation. Fig 3 shows the six effect predicates that result from Figure 1.

\[
\begin{align*}
E_1 : & \quad C_{10} \land C_{11} \land C_{12} \land \text{prd}(\text{Sub}_1) \land (C_6 \lor C_7) \land \text{prd}(\text{Sub}_6), \\
E_2 : & \quad C_{10} \land C_{11} \land C_{12} \land \text{prd}(\text{Sub}_1) \land \neg C_6 \land \neg C_7 \land \text{prd}(\text{skip}), \\
E_3 : & \quad (\neg C_{10} \lor \neg C_{11} \lor \neg C_{12}) \land \text{prd}(\text{Sub}_2) \\
& \quad \land C_2 \land C_3 \land C_4 \land \text{prd}(\text{Sub}_3) \\
& \quad \land (C_6 \lor C_7) \land \text{prd}(\text{Sub}_6), \\
E_4 : & \quad (\neg C_{10} \lor \neg C_{11} \lor \neg C_{12}) \land \text{prd}(\text{Sub}_2) \\
& \quad \land C_2 \land C_3 \land C_4 \land \text{prd}(\text{Sub}_3) \\
& \quad \land \neg C_6 \land \neg C_7 \land \text{prd}(\text{skip}), \\
E_5 : & \quad (\neg C_{10} \lor \neg C_{11} \lor \neg C_{12}) \land \text{prd}(\text{Sub}_2) \\
& \quad \land C_5 \land \text{prd}(\text{Sub}_4) \land \text{prd}(\text{Sub}_5) \\
& \quad \land (C_6 \lor C_7) \land \text{prd}(\text{Sub}_6), \\
E_6 : & \quad (\neg C_{10} \lor \neg C_{11} \lor \neg C_{12}) \land \text{prd}(\text{Sub}_2) \\
& \quad \land C_5 \land \text{prd}(\text{Sub}_4) \land \text{prd}(\text{Sub}_5) \\
& \quad \land \neg C_6 \land \neg C_7 \land \text{prd}(\text{skip}),
\end{align*}
\]

Fig. 3. Effect Predicates from the Example in Fig 1

Some of these effect predicates may not be satisfiable. For example, if \( C_2 \) and \( \neg C_6 \) were contradictory, then effect E4 would be unsatisfiable, which would mean it was not a possible behavior of the original operation. To avoid generating tests from such effects, an effect predicate, \( E_i \), is deleted if \( \text{Inv} \land \text{Pre} \land E_i \) is unsatisfiable (where \( \text{Pre} \)
is the precondition of the operation and \( \text{Inv} \) is the invariant and context information of the formal model). This satisfiability checking is decidable because all data structures - i.e. given sets - are finite.

Note that some effect predicates may be satisfiable, but still not reachable, because the states that satisfy \( \text{Inv} \wedge \text{Pre} \wedge E_i \) are not reachable by any sequence of operations. This can happen when the invariant is not the strongest possible invariant. Non-reachability of effects cannot be checked locally, since it is a global property of the system. This is one example of how test case generation can expose problems in the specification, even before the tests are run.

The computation of the effect predicates from the formal model is similar to slicing techniques, particularly to conditioned slicing [18] and to dynamic specification-based slicing [19]. One difference is that the formal model has to deal with parallelism.

3 Control Flow Coverage Criteria

Control flow coverage criteria [1] are widely used in structural or code-based software testing practice. A wide range of different criteria, based mainly on the structure of the control flow graph of programs [1, 20], help to select test suites or to measure their quality. The next two subsections apply several classical notions of coverage criteria to the above specification model - first for control-flow paths, then for the more detailed case where a decision contains multiple conditions.

3.1 Path Coverage

Many of the traditional code-based coverage criteria are focused on ensuring good coverage of loop constructs. There are no loops in B abstract machines, so criteria like linear code sequence and jump (LCSAJ) and the test effectiveness ratio \( \text{TER}_i \) hierarchy (for \( i > 2 \)) are not relevant. The most relevant coverage criteria are:

**Statement Coverage (SC):** the test set must execute every reachable statement.

**Decision Coverage (DC):** each decision is made true by some tests, and false by other tests. Decisions are the branch criteria which modify the flow of control in if-then-else and selection statements etc.

**Path Coverage (PC):** every satisfiable path through the control-flow graph is executed.

As noted in the testing literature [2, 1], for code-based coverage

\[ PC \Rightarrow DC \Rightarrow SC \]

In fact, with the restricted and deterministic control-flow graphs used in this paper, statement coverage and decision coverage are equal, because every arc out of every choice node contains a statement (either a simple substitution, or a \textit{skip} statement). For example, the \( \texttt{IF } C \texttt{ THEN } S \texttt{ END} \) construct in B is translated to

\[ C \wedge \text{prd}(S) \parallel \neg C \wedge \text{prd}(\text{skip}) \]

because the \( \text{prd}(\text{skip}) \) gives equalities like \( x' = x \). It is still possible to achieve decision coverage, without covering every path, so path coverage is more demanding than
decision and statement coverage. This means that the following relationship holds for effect predicate coverage

\[ PC \Rightarrow DC = SC \]

Path coverage is generally impossible to achieve in code-based testing, because the presence of loops usually gives an infinite number of paths. However, here there is a finite set of paths, exactly corresponding to the set of satisfiable effect predicates. This means that if every effect predicate is tested, path coverage is obtained.

The use of EDNF in this paper is similar to the commonly-used disjunctive normal form in previous work [4] but contains much fewer alternatives than usual, because only the control flow choice operators generate alternatives. It can still be exponential in size when there are \( P \) parallel operators, all containing \( N \) choices (giving \( N^P \) effects).

\((S_1 \[\ldots \[ S_N \] \] \land \ldots \land (S'_1 \[\ldots \[ S'_N \] \] )\)

In our industrial experience (six applications with over a hundred pages of B) this pattern does not arise often in operations. More typically, there are deeply nested choice constructs and the parallelism usually occurs at leaves or with a simple substitution as one argument. Furthermore, when a large number of EDNF predicates are generated, typically a lot are unsatisfiable, so they can be removed. For example, in an application involving validation of bank card transactions, one operation generated 6095 EDNF predicates, but only 649 were satisfiable. It is this set of satisfiable paths that give path coverage.

The above criteria view each decision as an atomic choice, but in practice decisions are complex predicates constructed with \( \land, \lor \) and \( \neg \) operators, combining primitive conditions. Exposing the internal structure of these decisions leads to an extended family of coverage criteria. This issue of how to treat multiple conditions without exponential test case explosion is a key point for test generation, and is discussed in detail in the next section.

### 3.2 Multiple Condition Coverage Criteria

Several structural coverage criteria for decisions with multiple conditions have been defined in the testing literature (Figure 4). Brief informal definitions are given here, but more details and formal definitions in Z are available elsewhere [20]. Note the terminology: a decision contains one or more primitive conditions, combined by disjunction, conjunction and negation operators.

**Condition Coverage (CC):** A test set achieves CC when each condition in the program is tested with a true result, and also with a false result. For a decision containing \( N \) conditions, two tests can be sufficient to achieve CC (one test with all conditions true, one with them all false), but dependencies between the conditions typically require several more tests.

**Decision/Condition Coverage (D/CC):** A test set achieves D/CC when it achieves both decision coverage (DC) and CC.
Fig. 4. The Hierarchy of Control-Flow Coverage Criteria for Multiple Conditions. $A \rightarrow B$ means that criteria $A$ is stronger than criteria $B$.

**Full Predicate Coverage (FPC):** A test set achieves FPC when each condition in the program is forced to true and to false, in a scenario where that condition is *directly correlated* with the outcome of the decision. A condition $c$ is directly correlated with its decision $d$ when either $d \Leftrightarrow c$ holds, or $d \Leftrightarrow \neg c$ holds [21]. For a decision containing $N$ conditions, a maximum of $2N$ tests are required to achieve FPC.

**Modified Condition/Decision Coverage (MC/DC):** This strengthens the *directly correlated* requirement of FPC by requiring the condition $c$ to *independently affect* the outcome of the decision $d$. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible conditions [22]. Achieving MC/DC may require more tests than FPC, but the number of tests generated is generally linear in the number of conditions.

**Multiple Condition Coverage (MCC):** A test set achieves MCC if it exercises all possible combinations of condition outcomes in each decision. This requires up to $2^N$ tests for a decision with $N$ conditions, so is practical only for simple decisions.

This section defines several rewrite rules, which split each effect predicate into several effect predicates, to satisfy more demanding coverage criteria. Recall that branches in each conditional statement are mutually exclusive. This means that only positive cases need to be considered in the rewrite rules, because negative cases are satisfied when another branch is chosen. The key issue is how disjunctions are handled within decisions. A simplistic way of viewing this is to consider a single disjunction $A \lor B$ nested somewhere inside a decision. There are four possible rewrite rules to transform the disjunction into a set of tests:

1. $A \lor B \leadsto \{ A \lor B \}$: This generates just one test for the whole disjunct, resulting in one test for the whole decision. This corresponds to decision coverage (because the negated decision is tested in another effect predicate).

2. $A \lor B \leadsto \{ A, B \}$: This ensures D/CC, because there is one test with $A$ true, and one test with $B$ true, and another effect predicate with the negated decision will test $\neg A \land \neg B$. In fact, a single test, $A \land B$, would in theory be enough to ensure D/CC, but $A \land B$ is often not satisfiable, so two weaker tests are generated instead.
3. $A \lor B \sim \{ A \land \neg B, \neg A \land B \}$: This is similar to FPC, because the result of the true disjunct is directly correlated with the result of the whole disjunction, since it cannot be masked by the other disjunct becoming true.

4. $A \lor B \sim \{ A \land \neg B, \neg A \land B, A \land B \}$: This corresponds to MCC, because it tries all combinations of $A$ and $B$ (the $\neg A \land \neg B$ combination will be tested in another effect predicate with the negated decision).

The next question is how these rewrite rules should be combined to act on the whole decision, and on the whole effect predicate (which generally contains a series of decisions). Using the usual distributive laws to propagate these tests up to the top level is a bad solution, because it is exponential and is not necessary to satisfy the coverage criteria. For example, an effect predicate that contained the following three decisions

\[
(A \lor B \land C) \land (D \lor E \lor F) \land (G \lor H)
\]

would generate $3 \times 3 \times 2 = 18$ predicates. Instead, each test is propagated up independently, which gives just $3+3+2=8$ predicates. For example, when $A$ is propagated up, other decisions remain unchanged, giving:

\[
A \land (D \lor E \lor F) \land (G \lor H)
\]

The Prolog algorithm shown in Figure 5 implements this strategy, using the second rewrite rule above. It is invoked by calling `dcc(Effect, Test)`, and it returns a sequence of tests (via backtracking) in the `Test` variable. For simplicity it is assumed that `Effect` contains just disjunction and conjunction operators, with all negation operators appearing at the leaves, and that the atomic conditions are just Prolog constants (atoms) — in practice, conditions may be any primitive predicates such as equalities, memberships etc. Note that `\+` is the standard Prolog negation operator.

This code is executable using most Prolog systems, as written, so the reader may wish to experiment with its behavior. It returns exactly one solution for each disjunct in `Effect`. The set of generated tests satisfies the D/CC criteria, since each condition is forced to be true, and another effect predicate with the negated decision will force each condition to be false. For example,

```prolog
?- dcc((a or b or c) & subs1 & (d or e or f) & subs2, Test).
Test = a \& subs1 \& (d or e or f) \& subs2 ;
Test = b \& subs1 \& (d or e or f) \& subs2 ;
Test = c \& subs1 \& (d or e or f) \& subs2 ;
Test = (a or b or c) \& subs1 \& d \& subs2 ;
Test = (a or b or c) \& subs1 \& e \& subs2 ;
Test = (a or b or c) \& subs1 \& f \& subs2 ;
No more solutions
```

The FPC criterion ensures that each condition is forced to be true at least once, and false at least once. The total number of tests generated for a decision containing $N$ conditions is $O(N)$. The Prolog algorithm for FPC is identical to the `dcc` code, except that clauses 4 and 5 return an additional conjunction:
\[ \text{:- op(800, xfy, \{\&\}).} \]
\[ \text{:- op(900, xfy, \{or\}).} \]
\[ \text{dcc(P \& Q, P2 \& Q) :- containsOR(P), dcc(P,P2).} \]
\[ \text{dcc(P \& Q, P \& Q2) :- containsOR(Q), dcc(Q,Q2).} \]
\[ \text{dcc(P \& Q, P \& Q) :- \& containsOR(P), \& containsOR(Q).} \]
\[ \text{dcc(P or _, P2) :- dcc(P,P2).} \]
\[ \text{dcc(_, or Q, Q2) :- dcc(Q,Q2).} \]
\[ \text{dcc(not(P), not(P)) :- atom(P).} \]
\[ \text{dcc(P, P) :- atom(P).} \]

% containsOR(Pred) is true iff Pred contains an ‘or’.
containsOR(_ or _). 
containsOR(P & _) :- containsOR(P),!.
containsOR(_ & Q) :- containsOR(Q),!.

\[ \text{Fig. 5. Prolog Algorithm to Generate Tests with D/CC Coverage} \]

\[ \text{fpc(P or Q, P2 \& not(Q)) :- fpc(P,P2).} \]
\[ \text{fpc(P or Q, not(P) \& Q2) :- fpc(Q,Q2).} \]

This gives the same number of solutions as the \text{dcc} algorithm \((O(N))\), but the solutions are more specific, to ensure that \(a\) is not masked by \(b\) or \(c\), for example. For each decision, this \text{fpc} algorithm generates the minimum number of tests required to achieve the FPC criteria. However, since it treats each decision independently (generating FPC tests from one decision at a time), it is sometimes possible to optimize the complete set of tests by merging tests from different decisions. For example, if \((a \text{ or } b \text{ or } c)\) and \((d \text{ or } e \text{ or } f)\) are separate decisions, then the set \(\{a \land d, b \land e, c \land f\}\), together with some negated tests, would achieve full predicate coverage \text{if these pairs were satisfiable}. But the above algorithm generates at most \(O(N)\) tests and places the minimal constraints on satisfiability that are possible, so it is more useful in practice.

For MCC, a similar Prolog algorithm can be used, considering each decision independently, but applying rewrite rule 4 to each disjunct. In the worst case, a decision with \(N\) disjunct conditions gives \(2^{N-1}\) predicates. However, this can still be practical for operations with complex control structures but simple decisions, because each decision is treated independently. For example, if an effect predicate contains \(D\) decisions, each containing \(N\) disjuncts, \(D \ast (2^{N-1})\) predicates are generated. This is much less than the complete DNF form, which would contain \(2^{D \ast (N-1)}\) predicates.

Table 1 are the results of applying each algorithm to the 6 EDNF effect predicates from Figure 1, plus several general cases to illustrate the complexity of the algorithms. The DC column (decision coverage) is included to show that one test per decision is sufficient to achieve decision coverage (and statement coverage), because of the all-paths coverage property of the set of EDNF predicates (Section 3.1).
### Table 1. The number of tests generated by each kind of coverage criteria

<table>
<thead>
<tr>
<th>Predicate</th>
<th>DC</th>
<th>D/CC</th>
<th>FPC</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>E4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>E5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>E6</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>One decision with N disjuncts</td>
<td>1</td>
<td>N</td>
<td>N</td>
<td>$2^N - 1$</td>
</tr>
<tr>
<td>Random term with N conditions</td>
<td>1</td>
<td>$N/2$</td>
<td>$N/2$</td>
<td>$(2^N)/2$</td>
</tr>
<tr>
<td>D decisions (conjoined), each with N disjuncts</td>
<td>1</td>
<td>$D \times N$</td>
<td>$D \times N$</td>
<td>$D \times (2^N - 1)$</td>
</tr>
</tbody>
</table>

#### 4 A New Family of Data-Oriented Coverage Criteria: Boundary Coverage

Structure-based coverage criteria have been thoroughly investigated in the testing literature for both control-flow and data-flow. However, data-oriented coverage criteria are less mature, and are more difficult to define because of the wide variety of data types and the huge state spaces.

This section develops a family of boundary testing coverage criteria. The underlying assumption is that there is a large set of possible inputs for each effect predicate, and that the behavior of the effect is relatively uniform within that set, so errors are more likely to be detected by testing the boundaries of the input set than interior points. This is the same assumption as the fault model of domain testing [23] [24, Chap. 6].

#### 4.1 Minima, Maxima and Ordering Functions

Let $x$ be a vector of variables whose values come from a state space $S$, and $\text{ord} : S \rightarrow T$ be an ordering function which maps each value of $S$ to some totally-ordered set $T$.

In most of the examples, $T$ is the set of integers, but order functions that return sequences of integers (ordered lexicographically), or real numbers are also useful. This allows us to define the set of maximum values of $S$, with respect to the ordering function, and similarly for the set of minimum values.

**Definition 1** Given a state space $S$, and an ordering function $\text{ord} : S \rightarrow T$, where $T$ is totally-ordered, the set of maximums and minimums of $S$ is defined as:

- $\max_{\text{ord}}(S) = \{ x : S \mid \exists y : S \cdot \text{ord}(y) > \text{ord}(x) \}$
- $\min_{\text{ord}}(S) = \{ x : S \mid \exists y : S \cdot \text{ord}(y) < \text{ord}(x) \}$

Note that there is not always a unique maximum (or minimum) element. As an extreme case, if the ordering function is the constant function which maps all elements
of $S$ to 0, then $\max(S) = S$. This ordering function can be useful for small state spaces where it is desirable to test every member.

A more typical example is $S = \mathbb{P}\{2, 4, 8\}$ ordered by set cardinality. This gives

$$\min(\mathbb{P}\{2, 4, 8\}) = \{\emptyset\}$$
$$\max(\mathbb{P}\{2, 4, 8\}) = \{\{2, 4, 8\}\}$$

However, if the state space is restricted to $S_2 := \{s : \mathbb{P}\{2, 4, 8\} \mid \#s \neq 3\}$ then there are three maxima (because they all evaluate to 2):

$$\max(S_2) = \{\{2, 4\}, \{4, 8\}, \{2, 8\}\}$$

An even more precise ordering for this $S_2$ state space would be

$$\text{ord} := (\lambda s : S_2 \bullet (\#s, \sum_{i \in s} i))$$  \hspace{1cm} \text{(ordered lexicographically)}$$

which considers the sum of the members. This gives a unique $\max_{\text{ord}}(S_2) = \{\{4, 8\}\}$.

Typically, this minimization and maximization process is applied to effect predicates (Section 2), which are rich state spaces defined over the cartesian product of some variables, plus predicates to restrict the possible states. This means that the minimization and maximization take into account the relationships between variables. This results in fewer solutions (and more precise/satisfiable solutions) than the simple cartesian product of the minima and maxima of the individual variables, which is well-known to produce many irrelevant tests [24, p161].

For example, given the state space $S_{xyz} := \{x, y, z : \mathbb{P}(1..4) \mid P\}$, where the predicate $P$ is

$$x \cup y \cup z \subset 1..4 \land$$
$$\text{disjoint}(x, y, z) \land$$
$$\#y \leq 1 \land \#z \leq 1 \land$$
$$(z = \{\} \Rightarrow y = \{\})$$

the ordering function $(\lambda x, y, z : \mathbb{P}(1..4) \bullet \#x + \#y + \#z)$, gives:

$$\min(S_{xyz}) = \{\{\}, \{\}, \{\}\}$$
$$\max(S_{xyz}) = \{x, y, z : \mathbb{P}(1..4) \mid P \land \#x = 3 \land \#y = 0 \land \#z = 0\} \cup$$
$$\{x, y, z : \mathbb{P}(1..4) \mid P \land \#x = 2 \land \#y = 0 \land \#z = 1\} \cup$$
$$\{x, y, z : \mathbb{P}(1..4) \mid P \land \#x = 1 \land \#y = 1 \land \#z = 1\}$$

The three sets that make up $\max$ contain respectively 4, 12 and 24 specific solutions. This can be reduced by using a more precise ordering function, such as ordering by the sum of the contents of each set, as well as the cardinality. An even more precise approach would be to rank the variables ($x$ then $y$ then $z$), which would give a unique maximum $\{(2, 3, 4), \{\}, \{\}\}$.

The above ordering functions were all based on the type of the variables. Our experience on industry examples has shown that simple and reasonably effective ordering functions can be chosen for each B data type (sets, relations, functions, sequences and integers). But more sophisticated ordering functions could take the structure or semantics of the predicate into account.
4.2 Boundary Coverage Criteria

This section takes an abstract view of tests. A test \( t \) is simply a value within the state space, \( t \in S \). Now the concepts of weak and strong boundary testing are defined.

**Definition 2** A set of tests, \( T \subseteq S \) satisfies weak boundary coverage with respect to an ordering function \( \text{ord} \) iff \( T \) includes at least one maximum of \( S \) and at least one minimum of \( S \). That is, iff:

\[
\min_{\text{ord}}(S) \cap T \neq \emptyset \land \max_{\text{ord}}(S) \cap T \neq \emptyset
\]

**Definition 3** A set of tests, \( T \subseteq S \), where \( S \) is non-empty, satisfies strong boundary coverage with respect to an ordering function \( \text{ord} \) iff every boundary value of \( S \) is in \( T \). That is, iff:

\[
\min_{\text{ord}}(S) \subseteq T \land \max_{\text{ord}}(S) \subseteq T
\]

Note that strong boundary testing implies weak boundary testing.

The state space of an effect predicate \( E \) is defined by a complex predicate over all the before and after state variables as well as the input and output parameters:

\[
Pre(s, i) \land Inv(s) \land Inv(s') \land E(s, i, s', o)
\]

where \( Pre \) is the precondition of the operation, \( s : S \) represents the before-state variables, \( s' : S \) represents the after-state variables, \( i : I \) is the set of input parameters, \( o : O \) is the set of output parameters. Weak or strong boundary coverage can be applied to any of these four sets of variables, or even to the union of several of them. However, for a given effect predicate \( E \), there are two subsets that are particularly interesting:

- The before-states that enable this effect predicate (that is, satisfy its precondition). These are interesting because it is necessary to reach one of these states (by invoking a sequence of operations of the system under test) before one can test this effect.
- The input parameters of the operation. These are interesting because for each effect of an operation, the invocation is realized at extremum values of the input variables domain.

Boundary coverage criteria are defined over these two sets of variables:

**Definition 4** A test set \( T \) achieves weak (strong) before-state boundary coverage of an operation \( Op \), with effect predicates \( E_1 \ldots E_n \), iff it achieves weak (strong) boundary coverage of the before-state \( BS \) of every effect predicate \( E_j \), where

\[
BS = \bigcup_{j=1}^n \{ s, s' : S ; i : I ; o : O \mid Pre(s, i) \land Inv(s) \land E_j(s, i, s', o) \bullet s \}
\]

**Definition 5** A test set \( T \) achieves weak (strong) input boundary coverage of an operation \( Op \), with effect predicates \( E_1 \ldots E_n \), iff it achieves weak (strong) boundary coverage of the input state \( IS \) of every effect predicate \( E_j \), where

\[
IS = \bigcup_{j=1}^n \{ s, s' : S ; i : I ; o : O \mid Pre(s, i) \land Inv(s) \land E_j(s, i, s', o) \bullet i \}
\]

Often, an effect predicate \( E \) works on a subset of the state variables, so maximizing over all the state variables creates an unnecessarily large set of maxima. In this case, a useful heuristic is to minimize and maximize only over the relevant variables of effect \( E \), which are the variables that it explicitly manipulates (reads or writes). This is done simply by hiding existentially all other variables. This is a strategy that has been found to reduce the number of test cases without significantly reducing the quality of tests [25].
5 Overview of the BZ-TT test generation method

This section presents the overall test generation process followed by BZ-TT. More detail is given elsewhere [9, 11]. After the formal model is written in B or Z, then translated to BZP format (before-after predicates with the $\mathbb{R}$ operator), there are three main phases:

1. **Test-Objective Generation:** generates test objectives (boundary goals) from the formal model of each operation.
2. **Test Construction:** converts each test objective into a test case (a sequence of abstract operation calls).
3. **Test reification:** transforms each test case into an executable test script, using a reification relationship between abstract test cases and concrete test scripts.

This paper focuses mostly on the first phase. For the issues of sequencing (Phase 2) see [26] or for reification (Phase 3) see [14].

Each test invocation is generated via the following steps:

1. Transformation of the postcondition of each operation into EDNF effect predicates, as described in Section 2, discarding unsatisfiable effect predicates. The goal of this is to achieve path coverage of the operation.
2. Transformation of each EDNF effect into one or several more detailed effect predicates using a coverage criteria algorithm such as D/CC, FPC or MCC. This is an optional step, which can be used when a particular kind of coverage is desired.
3. Calculation of one or more boundary goals from the before-state of each effect predicate. This process is controlled by the choice of ordering function, plus the choice between weak and strong before-state boundary coverage.

Each resulting boundary goal is a subset of the state space, represented by a set of constraints. The test construction phase of BZ-TT then tries to find a sequence of operations (the preamble) that reaches a boundary state, which is any state that satisfies the boundary goal. Unreachable boundary goals are discarded at this stage [9]. At this boundary state, one or more boundary values are chosen for the input variables of the operation invocation (the body). This achieves weak or strong input boundary coverage.

Every test case includes oracle checks on the outputs of each operation invocation, to check that the outputs agree with the expected state.

6 Controlling Test Case Explosion

The partial formal model of the system under test, developed specifically for testing purposes, defines the high-level testing objectives. It determines the abstraction level, which operations will be tested, which state variables are relevant, which inputs will be tested and which outputs will be observed. Our experience has shown that this style of focused model allows effective testing with fewer problems of test case explosion or excessive test computation time than a general model of all the system functionality. This point is also noted in [27].

The approximate number of tests generated is given by the following formula:

\[ \text{Tests} = \text{Ops} \times \text{Effs} \times \text{Control} \times \text{Boundaries}(V) \times \text{Boundaries}(I) \]
where \( \text{Ops} \) is the number of operations, \( \text{Effs} \) is the average number of effects per operation, \( \text{Control} \), \( \text{Boundaries}(V) \) and \( \text{Boundaries}(I) \) are as defined below (\( V \) stands for the number of state variables and \( I \) stands for the average number of input variables per operation). The formula is approximate, because the exact result depends upon how many of the tests are satisfiable and reachable and how many duplicate boundary goals are generated (duplicates are discarded). Also, a more accurate estimate can be obtained by applying the formula to each operation separately (with more precise statistics about that operation) and summing the results.

Note that the formal model determines \( \text{Ops} \) and \( \text{Effs} \). The test engineer has control over the other parameters during the generation process, with the default settings being \( \text{Control} = 1, \text{Boundaries}(V) = 2, \text{Boundaries}(I) = 2 \). So the default number of tests is four times the total number of effect predicates. Each of these controls can be set for the whole specification, then overridden for each operation, for each effect predicate, or even for each boundary goal.

1. The strategy for handling multiple conditions within decisions. Assume that an effect predicate contains \( D \) decisions, each with \( C \) conditions, of which \( J \) are disjuncts. Then the engineer can choose between the following coverage criteria:
   - DC: \( \text{Control} = O(1) \) tests per satisfiable and reachable effect predicate.
   - D/CC: \( \text{Control} = O(D \times J) \) tests.
   - FPC: \( \text{Control} = O(D \times J) \) tests.
   - MCC: \( \text{Control} = O(D \times (2^J - 1)) \) tests.

2. The ordering function. For each data type (or at a finer level, for each variable), an ordering function can be chosen from a standard library [9]. For example, this allows one to choose whether the members of an enumerated type should be viewed as uniform (tests will be generated for each member) or ordered (tests will focus on the first and last members only). Integer variables are usually ordered on their value, which is a total ordering, and thus has a unique minimum and maximum. Set variables are usually ordered on their cardinality, followed by the sum of their contents.

3. Strong versus Weak boundary testing. That is, how many maximal and minimal solutions to choose. Typically, with a given ordering function, many different test vectors may evaluate to the same value. This means there are many maximal values for that effect predicate. For a given vector of variables \( V \), the engineer can choose between:
   - weak boundary coverage: just one maximal and one minimal test vector are used as boundary goals (\( \text{Boundaries}(V) = 2 \)). In other words, minimization and maximization are done just once for each effect predicate.
   - N-dimensional weak boundary coverage: this minimizes and maximizes one time for each variable \( x \), giving that variable a higher priority during minimization and maximization. This is done by changing the ordering function \( \text{ord}(V) \) to \( \langle f(x) \rangle \wedge \text{ord}(V) \), where \( f \) is the ordering function for the variable \( x \). The effect is to treat each dimension of the N-dimensional state space separately, which results in a linear number of boundary points being found (\( \text{Boundaries}(V) = 2 \times V \)). For example, Figure 6 shows the effect of applying N-dimensional weak boundary coverage to a simple two-dimensional
Fig. 6. Weak boundary coverage (with ordering function \( (x + y) \)) compared with N-dimensional weak boundary coverage.

geometric figure—a minimum and maximum are obtained along each of the X and Y axes (using ordering functions \((x, x + y)\) and \((y, x + y)\), respectively), rather than just a single maximum and minimum. (This example is a typical domain testing problem, so the edge strategy discussed in Section 4.1 would be an even better way of testing each segment of the boundary).

- **strong boundary coverage:** all maximal and all minimal test vectors are used as boundary goals (Boundaries\((V)\) is \(O(M^V)\), where \(M\) is the average number of maximums per data type, and \(V\) is the number of variables). This is useful only when there are very few variables, or when strong ordering functions are used on all variables, so that \(M\) is small.

7 Example and Experiments

This section shows how the various test generation options work on a small example, and gives some general results about their application on large industrial case studies.

7.1 The Triangle Example

Figure 7 shows a B specification of the classic triangle example [2]. For test generation purposes, MAXSIZE is set to 10. This machine has no state variables, so test generation proceeds directly to analysis of the input variables. After transforming the operation to EDNF, the following four effect predicates are obtained (DNF would have given 36 predicates):

\[
E1 : (s1 + s2 \leq s3 \lor s2 + s3 \leq s1 \lor s1 + s3 \leq s2) \land \text{kind} = \text{invalid}
\]
\[
E2 : (s1 + s2 > s3 \land s2 + s3 > s1 \land s1 + s3 > s2) \land (s1 = s2 \land s2 = s3) \\
\land \text{kind} = \text{equilateral}
\]
\[
E3 : (s1 + s2 > s3 \land s2 + s3 > s1 \land s1 + s3 > s2) \land (s1 \neq s2 \lor s2 \neq s3) \\
\land (s1 = s2 \lor s2 = s3 \lor s3 = s1) \land \text{kind} = \text{isosceles}
\]
\[
E4 : (s1 + s2 > s3 \land s2 + s3 > s1 \land s1 + s3 > s2) \land (s1 \neq s2 \lor s2 \neq s3) \\
\land (s1 \neq s2 \land s2 \neq s3 \land s3 \neq s1) \land \text{kind} = \text{scalene}
\]
MACHINE
   TRIANGLE
SETS
   KIND = { scalene, isosceles, equilateral, invalid }
CONSTANTS
   MAXSIZE
PROPERTIES
   MAXSIZE = 10
OPERATIONS
   kind ← classify ( s1, s2, s3 ) =
      PRE s1 : 1..MAXSIZE ∧
              s2 : 1..MAXSIZE ∧
              s3 : 1..MAXSIZE ∧
      THEN
         IF s1 + s2 ≤ s3 ∨ s2 + s3 ≤ s1 ∨ s1 + s3 ≤ s2
            THEN kind := invalid
            ELSE
               IF s1 = s2 ∧ s2 = s3
               THEN kind := equilateral
               ELSE
                  IF s1 = s2 ∨ s2 = s3 ∨ s1 = s3
                     THEN kind := isosceles
                     ELSE kind := scalene
                  END
               END
         END
      END
END
END
END
END

Fig. 7. The Triangle specification in B

Table 2 shows the number of tests that result from applying various test generation control options to these four effects. Each column applies a different multiple-condition coverage algorithm, and the three groups of rows show the effects of selecting weak and strong boundary coverage, plus the intermediate option of N-dimensional boundaries.

Note that the FPC algorithm usually generates the same number of tests as D/CC, but because it generates tests that are more specific, some of them are unsatisfiable, which results in no tests for the scalene case (E4). This suggests that when a test generated by the FPC algorithm is inconsistent, one should fall back to using the corresponding D/CC test instead, to maintain at least that level of coverage.

To allow comparison, the actual boundary values produced by several of the more useful options are listed below:

DC Weak:  Invalid: (1,1,2), (10,9,1);
          Equilateral: (1,1,1), (10,10,10);
<table>
<thead>
<tr>
<th></th>
<th>Effect</th>
<th>DC</th>
<th>D/CC</th>
<th>FPC</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of satisfiable effects</td>
<td>E1(inv)</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>E2(equ)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>E3(iso)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>E4(sca)</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4</td>
<td>10</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Positive Tests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for 1 dimensional</td>
<td>E1(inv)</td>
<td>1.1</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
</tr>
<tr>
<td>weak boundary coverage</td>
<td>E2(equ)</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>E3(iso)</td>
<td>1.1</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>E4(sca)</td>
<td>1.1</td>
<td>1.1</td>
<td>0.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4.4</td>
<td>8.8</td>
<td>7.7</td>
<td>8.8</td>
</tr>
<tr>
<td>Positive Tests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for N dimensional</td>
<td>E1(inv)</td>
<td>2.3</td>
<td>3.6</td>
<td>3.6</td>
<td>3.6</td>
</tr>
<tr>
<td>weak boundary coverage</td>
<td>E2(equ)</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>E3(iso)</td>
<td>2.2</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>E4(sca)</td>
<td>3.3</td>
<td>3.3</td>
<td>0.0</td>
<td>3.3</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>8.9</td>
<td>10.13</td>
<td>7.10</td>
<td>10.13</td>
</tr>
<tr>
<td>Positive Tests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for 1 dimensional</td>
<td>E1(inv)</td>
<td>3.27</td>
<td>3.27</td>
<td>3.27</td>
<td>3.27</td>
</tr>
<tr>
<td>strong boundary coverage</td>
<td>E2(equ)</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>E3(iso)</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>E4(sca)</td>
<td>6.6</td>
<td>6.6</td>
<td>0.0</td>
<td>6.6</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>13.37</td>
<td>13.37</td>
<td>7.31</td>
<td>13.37</td>
</tr>
</tbody>
</table>

Table 2. Test Results for Triangle Classify Operation. Each $i,j$ entry represents the number of minimal ($i$) and maximal($j$) tests.
Isosceles: (1,2,2), (10,10,9);
Scalene: (2,3,4), (10,9,8).

**DC N-dim:**
Invalid: (1,1,2), (1,2,1), (10,9,1), (9,10,1), (9,1,10);
Equilateral: (1,1,1), (10,10,10);
Isosceles: (1,2,2), (2,1,2), (10,10,10), (10,9,10);
Scalene: (2,3,4), (3,2,4), (3,4,2), (10,9,8), (9,10,8), (9,8,10).

**D/CC Weak:**
Invalid: (1,1,2), (2,1,1), (1,2,1), (9,1,10), (10,9,1), (9,10,1);
Equilateral: (1,1,1), (10,10,10);
Isosceles: (1,2,2), (2,1,2), (2,2,1), (10,9,10), (10,10,9), (9,10,10);
Scalene: (2,3,4), (10,9,8).

### 7.2 Industrial Case Studies

The BZ-TT method and tool set has been developed since 1999 on the basis of several industrial case-studies. These applications of the automated test generation process has been carried out in partnership with two companies: SchlumbergerSema (two divisions: Smart Card RD which provides smart card software, and e-City which develop urban systems), and PSA Peugeot Citroën.

This section presents some coverage statistics from a Smart Card key management application. The formal model is more than 50 pages of B, with 11 operations which model the smart card commands. Table 3 gives statistics for two typical operations, showing the number of feasible effect predicates for each condition coverage criteria, and the resulting number of weak and strong boundary tests.

<table>
<thead>
<tr>
<th>Operations</th>
<th>DC</th>
<th>D/CC</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATEFLA</td>
<td>Feasible Effect Predicates Weak/Strong Boundary Tests</td>
<td>8/220</td>
<td>15/16/876</td>
</tr>
<tr>
<td>VERIFYPIN</td>
<td>Feasible Effect Predicates Weak/Strong Boundary Tests</td>
<td>22/14/2466</td>
<td>307/141/107</td>
</tr>
</tbody>
</table>

**Table 3.** Example Statistics for Smart Card Industrial Case Study

This illustrates that strong boundary coverage often produces too many tests to be practical. N-dimensional boundary coverage gives a number of tests in between weak and strong coverage, so can be a useful compromise.

### 8 Related Work

The research on Model-Based Automated Test Generation Tools is currently very active; see [28] for a review of different tools, both commercial and academic. These tools use as input a formal model of the system under test and allow the validation engineer to drive the test generation process.
Formal description languages based on Labelled Transition Systems (LTS), Extended Finite State Machines (EFSM) or Abstract State Machines (ASM) have been widely considered, see e.g. [29]. In these approaches, the test generator may support a variety of existing transition-based strategies like state coverage, path coverage, constrained path coverage, all transition pairs, etc. The main limitation of these approaches comes from the state explosion problem and the test case explosion problem [30]. Several proposals address these limitations, for example on the basis of the formalization of reachable properties guiding the test generation or by reducing the test suites using individual requirements selection [31].

Set-oriented model-based formal notations, like VDM, Z or B have been extensively studied for test generation purpose. Most approaches [4, 7, 6] use a partition analysis of the operation to build a Finite State Automaton - FSA - corresponding to an abstraction of the reachability graph denoted by the specification. Test cases are then generated using the same kinds of coverage criteria as used by the LTS/EFSM/ASM approaches, with the same limitations (test case explosion). Moreover, the transformation of the formal model into an abstract FSA introduces several fundamental problems such as the non-discovery problem and again the state explosion problem [29]. An other approach [32] introduces so-called Testgraph as test objective to drive the sequencing of the operation invocations.

The BZ-TESTING-TOOLS approach proposes another way, which is also based on the partition analysis of the operation, but avoids the a priori construction of the FSA. The test generation is then conducted by Boundary Goal calculation, guiding the computation of the preambles. Moreover, Constraint Logic Programming [10], used as a basis for animation purposes, allows reasoning on so-called constraint states, represented by constraint stores, which denote sets of valued states. This reduces the combinatorial explosion. The coverage criteria are then defined on the basis of a structural analysis of the B or Z model. Such an approach has already been considered by Behnia and Waeselynck [8] who study test criteria definition for B models. But they have a different purpose, which is to test the B formal model itself for validation purposes, starting from a B project multi-layered machine with refinement and implementation. Thus, they have to consider in the structural analysis, implementation structures like WHILE and sequencing. BZ-TT performs the analysis on abstract machines instead, where loops and sequencing are not allowed, which simplifies the analysis, makes complete path coverage feasible, and allows a focus on more sophisticated coverage criteria for decisions that contain multiple conditions.

Some other researchers have investigated specification-based coverage criteria in state-oriented specifications, like full-predicate, transition-pair and specification-mutation coverage [21, 33]. One way in which this paper extends that work, is by using full-predicate coverage in multiple condition analysis.

The second strategy used by BZ-TT is boundary testing. This is widely used as an informal heuristic during manual test design. A partially automated boundary test generation system is described in [34]. They define a family of boundary heuristics ($k$-bdy), where 1-bdy generates all combinations of maximum and minimum values of an N-dimensional integer input space. This roughly corresponds to the mixed boundary heuristic using all variables, for the special case of integer input domains (which are
totally ordered). However, an important difference is that they blindly generate all the boundary points, then discard those that are invalid (do not satisfy the precondition). This can result in many useful tests being missed, and could even result in zero valid test cases being generated. In contrast, in this paper the search for each boundary test case considers the precondition, which means that only valid test inputs are generated, thus obtaining much more precise coverage of the real (semantic) boundary points.

Hoffman et. al. [34] also define a family of perimeter strategies (k-per), where 1-per holds one variable at a boundary value but allows the others to vary. This is similar in philosophy to the strategy mentioned above of using a subset of the variables during boundary analysis. However, this subset is chosen by considering the semantics of the operation under test (which variables it modifies), whereas Hoffman just forces one variable at a time to have a boundary value.

9 Conclusions and Future Work

This paper has made advances in five points related to model-based black-box testing.

Firstly, we have described ways of analyzing B abstract machines in terms of control flow analysis. This allows specifications to be transformed into EDNF form, which is more practical than DNF analysis, and provides a connection to structure-based coverage criteria. A similar process can be followed for Z specifications, but the coverage results are less clear because Z specifications are less structured than B machines (where every condition has an associated substitution statement).

Secondly, we have adapted the classical structure-based control-flow coverage criteria to predicate-based specifications, particularly for multiple condition coverage criteria. We have defined algorithms which satisfy each of these coverage criteria in a practical fashion. They do not necessarily produce a minimal set of tests, because each decision is treated independently to minimize the possibility of generating unsatisfiable effect predicates (and thus losing coverage).

Thirdly, we have described a toolbox of techniques for reducing and controlling test case explosion, which is a crucial issue for the scalability of test generation. The classic DNF and FSA approach does not scale well. The techniques described in this paper for improving scalability include: using EDNF rather than DNF, computing the EDNF from each operation rather than the whole model, exploring the reachable states on the fly (guided by boundary goals) rather than constructing the whole FSA, and a systematic range of coverage controls which allow the test engineer to generate a predictable number of tests.

Fourthly, we have introduced a family of data-oriented boundary coverage criteria, parameterized by the ordering function. The calculation of boundaries uses the predicates of the specification, which gives very precise boundaries.

Finally, we have synthesized these coverage criteria into a practical sequence of control parameters that allows the test engineer to control test case explosion, with clear coverage consequences. These control parameters are embedded in the BZ-TT tool set and have been used in several industrial case studies. An academic version of this environment will be released in 2004.
These five points advance the state of the art in the area of model-based test generation from notations like B. Practical systems and techniques for model-based test generation will be a major improvement of current testing practice by making test generation more systematic and reducing costs.

The approach we have described uses before-after predicates to specify operations, with few restrictions on the structure of those predicates. Thus, it is general enough to be applied to many other specification notations. We have not addressed other issues of functional black-box testing such as reactive systems, concurrency and timing. However, we are adapting this approach to work with reactive systems modelled using statecharts.

Other future work will be to develop better reachability tests, better support the MC/DC coverage criteria, and better integrate domain testing strategies, like the edge strategy. On the tools side, more experience is needed to determine if the control parameters that we have proposed provide sufficient control over test case generation, and whether they are rich enough to allow test engineers to exercise their validation expertise.

Acknowledgments

We would like to thank Paul Strooper and Tim Miller from the University of Queensland and Alain Giorgetti from the University of Franche-Comté for useful feedbacks on an earlier draft of this paper. Thanks to the anonymous referees for many helpful comments.

References


