Boundary Coverage Criteria for Test Generation from Formal Models

Nikolai Kosmatov
Lehrstuhl D für Mathematik
Aachen, Germany
Email: kosmatov@lifc.univ-fcomte.fr

Bruno Legeard
Laboratoire d'Informatique
Université de Franche-Comté - CNRS - INRIA
16, route de Gray - 25030 Besançon, France
Email: legeard@lifc.univ-fcomte.fr

Fabien Peureux
Laboratoire d'Informatique
Université de Franche-Comté - CNRS - INRIA
16, route de Gray - 25030 Besançon, France
Email: peureux@lifc.univ-fcomte.fr

Mark Utting
Department of Computer Science
The University of Waikato
Private Bag 3105 - Hamilton, New-Zealand
Email: marku@cs.waikato.ac.nz

Abstract

This paper proposes a new family of model-based coverage criteria, based on formalizing boundary-value testing heuristics. The new criteria form a hierarchy of data-oriented coverage criteria, and can be applied to any formal notation that uses variables and values. They can be used either to measure the coverage of an existing test set, or to generate tests from a formal model. We give algorithms that can be used to generate tests that satisfy the criteria. These algorithms and criteria have been incorporated into the BZ-TESTING-TOOLS *(*BZ-TT*) tool-set for automated test case generation from B, Z and UML/OCL specifications, and have been used and validated on several industrial applications in the domain of critical software, particularly smart cards and transport systems.*

Keywords: *model-based testing, boundary-value testing, test coverage criteria*

1 Introduction

One major issue in automated model-based testing is to be able to measure and maximize coverage of the model. The intention is not to guarantee test coverage of the implementation – its structure may be quite different to that of the model. Rather, the formal model represents the high-level test objectives: it expresses the aspects of the system behavior that the engineer wants to test. So, covering all parts of those behaviors is important. Furthermore, good objective model-based coverage criteria are needed to give a reproducible and mature approach to black-box testing, and to determine when sufficient testing has been done.

Coverage criteria for model-based testing is a relatively new area of research. Various criteria have been proposed, based on the kind of specification notation used for the models or on adapting code-based coverage criteria. We briefly discuss each of these.

The style of specification notation naturally suggests several kinds of coverage:

- Transition notations, such as Statecharts[14] and finite state machines, lead to coverage criteria like *all transitions* and *all transition pairs* [24].
- ASM notation (Abstract State Machine) uses rules to describe behavior, so leads to a notion of *rule coverage* [13].
- State-based notations, such as Z [26], VDM [17] and B [1], where behavior is described by predicates, suggest coverage criteria based on the structure of the predicates, like DNF (disjunctive normal form) coverage [12].

Code-based coverage criteria [31], such as *statement coverage* (SC), *decision coverage* (DC), *decision/condition coverage* (D/CC), *modified condition/decision coverage* (MC/DC) [10], *path coverage* (PC), have been adapted to be applied to formal models [24, 20]. Some data-flow coverage criteria, like *definition-use coverage*, have also been applied to notations such as Statecharts [16].

On the other hand, classical informal testing heuristics [22], such as cause-effect analysis, partition analysis and boundary/domain testing [3], are currently used as the basis for test generation algorithms, but have not generally been formalized as coverage criteria.

This paper proposes a new family of model-based coverage criteria, by formalizing boundary testing heuristics. These new criteria can be used either to evaluate an existing test set, or to generate tests from a formal model. They are data-oriented criteria, so can be applied to any formal notation that uses variables and values.

Because they are data-oriented, they are largely independent of the structural coverage criteria such as transition coverage, rule coverage and DNF coverage. This is one of the key reasons that our new criteria are important – *after* structural criteria have been used to generate sets of test objectives (expressed as predicates), our new criteria can be used to obtain more precise coverage of each test objective, or more control over how many tests are generated for each test objective. So our new criteria complement and enhance the existing widely used criteria.

Most of these criteria are implemented in the BZ-TESTING-TOOLS (BZ-TT) environment [2, 8], where they can be used to control the generation of tests from B abstract machines [1], Z specifications [26] and UML class diagrams with OCL pre/post specifications of methods [25, 29]. The BZ-TT tool-set uses a customized set-oriented constraint solver that makes it possible to efficiently implement the boundary selection method. The use of the BZ-TT technology on industrial applications in the domain of smart card software (GSM 11-11 standard [18], Java Card transaction mechanism [5]) and transport applications (Metro/RER ticket validation algorithm [9], automobile windscreen-wiper controller) has shown the strength of the boundary approach for fault detection.

In previous work, we have presented the BZ-TT method to generate test cases using a boundary oriented approach from a formal model [19], to control the test case explosion [20] and to generate test drivers from test cases [5]. In this paper, we introduce and formalize the boundary coverage criteria which give the rational for the (BZ-TT) method and tool-set.

So, the paper is structured as follows. In Section 2, we introduce concepts relating to boundaries. In Section 3, we define the boundary coverage criteria and discuss the relations between different criteria. Our method of boundary state generation is presented in Section 4. We use the small classical triangle example to illustrate the test generation based on our criteria in Section 5. In Sections 6, 7 and 8, we respectively present related work, discuss the interests to formalize boundary coverage criteria and give the conclusions and future work. The proofs of our results are given in Annexe.

2 Formal Models and Boundary States

For generality, we take an abstract view of the notation used to describe formal models. We assume that the formal model is defined by state variables, an initial state, and a set of operations which may have input and output variables. The operations are described by predicates: an invariant predicate $\mathcal{I}$ that specifies the allowable values of the state variables, a precondition predicate and a postcondition predicate for each operation. Many different kinds of formal specification notation can be expressed or translated into this form, including B, Z, VDM, UML/OCL and ASM notations.

2.1 BZ-TT Partitioning through Small Example

The BZ-TT method consists of testing all the possible behaviors of the system when it is in a *boundary state* of its subdomains. This test generation process is highly automated and fully supported by the BZ-TT tool-set [19, 2].

A behavior of the system on the points of a subdomain is defined by a predicate which is called an *effect predicate*. One effect predicate defines one behavior of the specification. More precisely, an effect predicate contains two parts: the conditions on input and state variables defining the subdomain (before part), and the predicates defining the system behavior on this subdomain (after part). We include the precondition of the operation into the first part of the effect predicate.

In the BZ-TT approach, the first step consists in partitioning each operation of the formal model to compute these effect predicates. This partition analysis is similar to the DNF analysis introduced in [12]. An effect predicate corresponds to a disjunct of the whole formula for each operation. In fact, the formula, representing the DNF of the operation, is computed by introducing some optimizations to control the combinatorial explosion underlying to the DNF partitioning [7].

If the input formal model is a B abstract machine [1], then to compute the effect predicates, we translate the operations of the specification into before-after predicates. Basically, this partition analysis consists of unfolding predicates along branches, and introducing primed variables to denote the after values (the translation scheme from B generalized substitutions to before-after predicates is precisely defined in the B-Book [1, Chap. 6]).

Each computed before-after predicate constitutes one effect predicate. An effect predicate PS_i has the form

$$PS_i = \mathcal{I} \wedge Before_i \wedge After_i$$

where $\mathcal{I}$, $Before_i$ and $After_i$ respectively denote the invariant properties, the before and after predicates of the ef-

```
MACHINE
    $TRIANGLE$
SETS
    $KIND = \{scalene, isosceles, equilateral, invalid\}$
CONSTANTS
    $MAXSIZE$
PROPERTIES
    $MAXSIZE = 10$
OPERATIONS
$kind \longleftarrow classify(s1, s2, s3) =$
    PRE
        $s1 : 1 \,..\, MAXSIZE \wedge$
        $s2 : 1 \,..\, MAXSIZE \wedge$
        $s3 : 1 \,..\, MAXSIZE$
    THEN
        IF $s1 + s2 \leqslant s3 \vee s2 + s3 \leqslant s1 \vee s1 + s3 \leqslant s2$
        THEN $kind := invalid$
        ELSE
            IF $s1 = s2 \wedge s2 = s3$
            THEN $kind := equilateral$
            ELSE
                IF $s1 = s2 \vee s2 = s3 \vee s1 = s3$
                THEN $kind := isosceles$
                ELSE $kind := scalene$
                END
            END
        END
    END
END
```

Figure 1. The B abstract machine for the triangle specification

$$
\begin{array}{lcl}
PS_1 & : & Pre \wedge (s1 + s2 \leqslant s3) \wedge kind = invalid \\
PS_2 & : & Pre \wedge (s2 + s3 \leqslant s1) \wedge kind = invalid \\
PS_3 & : & Pre \wedge (s1 + s3 \leqslant s2) \wedge kind = invalid \\
PS_4 & : & Pre \wedge (s1 + s2 > s3) \wedge (s2 + s3 > s1) \wedge \\
 & & (s1 + s3 > s2) \wedge (s1 = s2) \wedge (s2 = s3) \wedge \\
 & & kind = equilateral \\
PS_5 & : & Pre \wedge (s1 + s2 > s3) \wedge (s2 + s3 > s1) \wedge \\
 & & (s1 + s3 > s2) \wedge (s1 \neq s2) \wedge (s2 = s3) \wedge \\
 & & kind = isosceles \\
PS_6 & : & Pre \wedge (s1 + s2 > s3) \wedge (s2 + s3 > s1) \wedge \\
 & & (s1 + s3 > s2) \wedge (s1 \neq s2) \wedge (s1 = s3) \wedge \\
 & & kind = isosceles \\
PS_7 & : & Pre \wedge (s1 + s2 > s3) \wedge (s2 + s3 > s1) \wedge \\
 & & (s1 + s3 > s2) \wedge (s2 \neq s3) \wedge (s1 = s2) \wedge \\
 & & kind = isosceles \\
PS_8 & : & Pre \wedge (s1 + s2 > s3) \wedge (s2 + s3 > s1) \wedge \\
 & & (s1 + s3 > s2) \wedge (s1 \neq s2) \wedge (s2 \neq s3) \wedge \\
 & & (s3 \neq s1) \wedge kind = scalene
\end{array}
$$

Figure 2. Effect predicates of the triangle example

fect i. The domain of the effect predicate PS_i is defined by $\mathcal{I} \wedge Before_i$.

Due to the lack of space, we do not present in this paper a full real-size application like in [4], but we illustrate the effect predicate computation and our coverage criteria with a B abstract machine of the triangle example [22].

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Fig. 1 shows a formal B specification of this example.

In the triangle example, there are no state variables and therefore no invariant properties. From the single operation *classify* of the triangle specification, we compute the 8 satisfiable effect predicates shown in Fig. 2 where

$$Pre = (s1 \in 1 \,..\, MAXSIZE) \wedge (s2 \in 1 \,..\, MAXSIZE) \\ \wedge (s3 \in 1 \,..\, MAXSIZE).$$

We recall that in B notation $s1 \in 1 \,..\, MAXSIZE$ means that $s1 \in \mathbb{Z}$ and $1 \leqslant s1 \leqslant MAXSIZE$. For example, in PS_1 we have:

$$Before_1 : Pre \wedge (s1 + s2 \leqslant s3), \\ After_1 : kind = invalid.$$

More generally, let $x_1, \dots, x_n$ be the (state and input) variables and I be the set of all the elements $(x_1, \dots, x_n)$ satisfying the invariant $\mathcal{I}$. If $\mathcal{C}$ is the Before part of an effect predicate, the set D defined as a subset of I by $\mathcal{C}$:

$$D = \{\, (x_1, \dots, x_n) \in I \mid (x_1, \dots, x_n) \text{ satisfies } \mathcal{C} \} \quad (1)$$

will be called the *domain* of this effect predicate. The boundary values of a predicate being defined by its domain, we can formulate our results in terms of domains. We assume that all our domains are bounded. For $D \subset \mathbb{Z}^n$ it is equivalent to saying that D is finite. Although our criteria and results make sense for closed continuous domains as well as for discrete ones, in this article we will be interested in the case of discrete domains only.

2.2 Definition of a Boundary State for a Discrete Domain

We first introduce the definitions for the continuous case. Let $D \subset \mathbb{R}^n$ be a closed domain of values of variables

$$x_1, \dots, x_n$$

in $\mathbb{R}$. An element

$$A = (a_1, \dots, a_n) \in D$$

is called *a boundary state of* D if for every $\varepsilon > 0$ the ball $B_\varepsilon(A)$ of radius ε and center a contains at least one point of $\mathbb{R}^n \setminus D$. The set of the boundary states of D is called *the boundary (frontier) of* D and denoted $\text{Fr}(D)$.

In case the variables $x_1, \dots, x_n$ take their values in $\mathbb{Z}$ and therefore D is a subset of $\mathbb{Z}^n$, we say that D is *a discrete domain*. If we consider $D \subset \mathbb{Z}^n$ as a subset of $\mathbb{R}^n$, D coincides with its boundary. The definition of a boundary state we have given above is not useful, because every element of D would be a boundary state. In practice, a discrete domain $D \subset \mathbb{Z}^n$ is often defined by an intermediate continuous domain: one first defines a real domain $D' \subset \mathbb{R}^n$ and puts $D = D' \cap \mathbb{Z}^n$. Intuitively, we would like to define a boundary state of a discrete domain D as a point of D near the boundary of D'. We proceed as for a continuous one. First of all, we define the discrete neighborhood of a point $A \in \mathbb{Z}^n$ as the set of its neighbor points. If all the neighbors of the point $A \in D$ are in D, A is obviously an interior point of D. So we say that a point of a discrete domain is a boundary state if some of its neighbors are outside the domain.

Let us now give the formal definitions. Let $A = (a_1, \dots, a_n)$ be a point of $\mathbb{Z}^n$. *The discrete neighborhood* $V(A)$ *of* A is the set

$$\begin{gathered} V(A) = \{\, A,\, (a_1 \pm 1, a_2, a_3, \dots, a_n), \\ (a_1, a_2 \pm 1, a_3, \dots, a_n), (a_1, a_2, a_3 \pm 1, \dots, a_n), \dots, \\ (a_1, a_2, a_3, \dots, a_n \pm 1)\,\}. \end{gathered} \tag{2}$$

In other words, $V(A)$ contains the points such that all their coordinates are equal to the corresponding coordinates of A except one which differs from the corresponding coordinate of A by at most 1. For example, Fig. 3a) shows the 5 points of $V((-3, 3))$ in $\mathbb{Z}^2$.

We can give an alternate definition of the discrete neighborhood $W(A)$:

$$\begin{gathered} W(A) = \{\, (b_1, \dots, b_n) \in \mathbb{Z}^n \mid \\ \forall\, i = 1, 2, \dots, n, \mid b_i - a_i \mid \leqslant 1 \,\}. \end{gathered} \tag{3}$$

$W(A)$ contains the points such that their coordinates differ from the corresponding coordinates of A by at most 1. We note that $A \in V(A) \subset W(A)$. Fig. 3b) shows the 9 points of the discrete neighborhood $W((-3, 3))$ in $\mathbb{Z}^2$.

Let $A = (a_1, \dots, a_n)$ be a point of a discrete domain D. We say that A is *a boundary state of* D if $V(A)$ contains at least one point of $\mathbb{Z}^n \setminus D$. *The boundary of* D is the set of its boundary states, it is denoted by $\text{Fr}(D)$.

To illustrate our definitions, we will consider the discrete domain D defined as in (1):

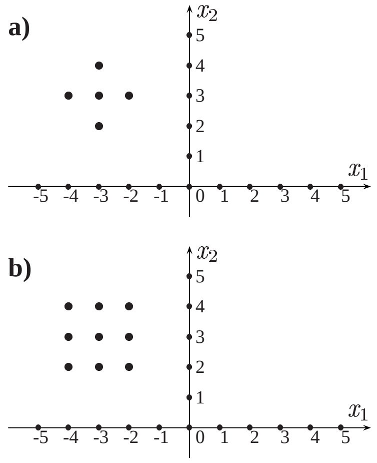

Figure 3. The discrete neighborhoods a) $V((-3, 3)) \subset \mathbb{Z}^2$**, b)** $W((-3, 3)) \subset \mathbb{Z}^2$**.**

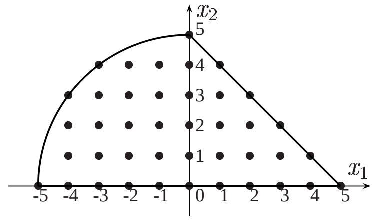

Figure 4. The discrete domain D **defined by (4).**

$$\begin{gathered} n = 2,\ \mathcal{C}_1 : x_1^2 + x_2^2 \leqslant 25, \\ \mathcal{C}_2 :\ x_1 + x_2 \leqslant 5,\ \mathcal{C}_3 :\ 0 \leqslant x_2, \\ D = \{\, (x_1, x_2) \in \mathbb{Z} \times \mathbb{Z} \mid \\ (x_1, x_2) \text{ satisfies } \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \}. \end{gathered} \tag{4}$$

Fig. 4 shows the 41 points of this domain. The 22 boundary states of D are shown on Fig. 5a). If we replace $V(A)$ by $W(A)$ in the definition of a boundary state, we will get 27 boundary states which are shown on Fig. 5b).

Both neighborhoods $V(A)$ and $W(A)$ seem equally interesting and either of them can be used in the definition of a boundary state. The choice depends on the kind of boundary we want to obtain. In the sequel, we will use only the discrete neighborhood $V(A)$. But our results will be obviously true if we replace it by $W(A)$, because $V(A) \subset W(A)$ and therefore using of $W(A)$ gives a "thicker" boundary.

It is clear that $\text{Fr}(D) \neq \varnothing$ for any domain $D \neq \varnothing$ as

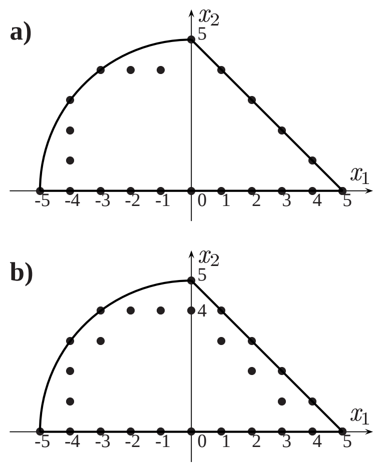

Figure 5. a) $\mathrm{Fr}(D)$. **b)** $\mathrm{Fr}(D)$ **defined using** $W(A)$ **rather than** $V(A)$**.**

all our domains are bounded. The reader will easily prove a stronger result:

Proposition 1 *Let* $D \neq \varnothing$ *be a domain of values of the variables* $x_1, \ldots, x_n$ *and* x_i *be one of these variables. Then there exist boundary states of* D *minimizing and maximizing* x_i *over* D*. Proof in Annexe (Section 10).*

2.3 Generalized Edges of a Domain

For better boundary testing, one may want to test each of the subconstraints defining our domain. For example, the domain D of (4) is defined by the conjunction of three constraints $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$. In this case we would like to test at least one point in every of the following subsets of $\mathrm{Fr}(D)$ that will be denoted respectively by E_1, E_2 and E_3:

These sets can be defined by $E_j = \mathrm{Fr}(D_j) \cap D$ where D_j is the domain defined by the constraint $\mathcal{C}_j$ only. The sets E_j will be called *generalized edges* of the domain. This terminology is motivated by the fact that if $D \subset \mathbb{R}^n$ is a closed convex polyhedron defined by linear inequalities $\mathcal{C}_j$, the sets $E_j = \mathrm{Fr}(D_j) \cap D$ are just the edges of D in usual sense.

For a formal definition, consider a domain D defined as in (1) by a conjunction of constraints

$$\mathcal{C} = \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_l, \quad l \geqslant 1. \tag{5}$$

Let $D_j \subset I$ be the domain defined by the constraint $\mathcal{C}_j$ only. Then it is clear that

$$D = D_1 \cap \ldots \cap D_l.$$

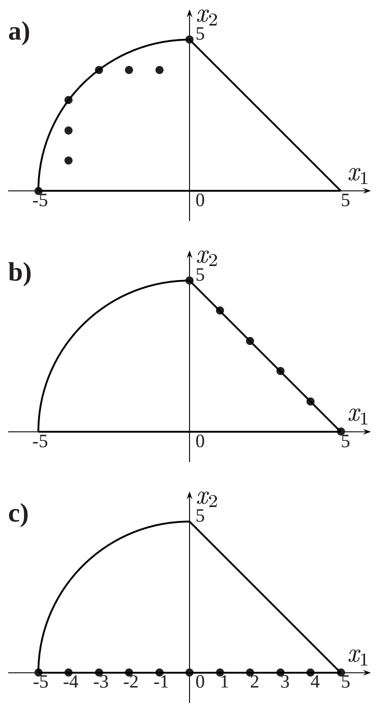

Figure 6. The edges a) E_1**, b)** E_2**, c)** E_3 **for the domain** D **defined by (4).**

The sets $E_j = \mathrm{Fr}(D_j) \cap D, \ j = 1, \ldots, l$, are called *the (generalized) edges of* D. It is easily proved that they compose $\mathrm{Fr}(D)$ (see Proof in Section 10):

$$\mathrm{Fr}(D) = \bigcup_{j=1}^{l} E_j. \tag{6}$$

In general, any non-empty domain D has non-empty edges by (6) as $\mathrm{Fr}(D) \neq \varnothing$. Some edges of D can be empty, for example, if we set

$$I = \mathbb{Z}, \ n = 1, \ l = 2, \ \mathcal{C}_1 : 0 \leqslant x \leqslant 25, \\ \mathcal{C}_2 : \ 2 \leqslant |\ x\ | \leqslant 8, \tag{7}$$

we have

$$D = \{x \in \mathbb{Z} \mid 2 \leqslant x \leqslant 8\}, \\ E_1 = \varnothing, \ E_2 = \mathrm{Fr}(D) = \{2, 8\}.$$

In most cases, for example, if all the $\mathcal{C}_j$ are linear inequalities, $E_1 = \varnothing$ means that the constraint $\mathcal{C}_1$ actually does not restrict the domain and

$$\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \ldots \wedge \mathcal{C}_l = \mathcal{C}_2 \wedge \ldots \wedge \mathcal{C}_l, \\ D = D_2 \cap \ldots \cap D_l$$

(note that this is not true in (7) where $\mathcal{C}_2$ is not a linear inequality). In any case, we could assume without loss of generality that such unnecessary constraints were dropped away one after another, and no other $\mathcal{C}_j$ can be dropped away from

$$\mathcal{C} = \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_l$$

without changing the domain D defined by $\mathcal{C}$.

In practice, the constraints $\mathcal{C}$ can be presented in the form (5) in different ways. For example, we could define the domain D of (4) by only two constraints:

$$\mathcal{C}'_1 = \mathcal{C}_1,\ \mathcal{C}'_2 = \mathcal{C}_2 \wedge \mathcal{C}_3$$

and consider the representation $\mathcal{C} = \mathcal{C}'_1 \wedge \mathcal{C}'_2$ rather than $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3$. As the objective of our edge criteria is to test one or several points on every edge E_j, it seems natural to take as $\mathcal{C}_j$ the "smallest" subconstraints of $\mathcal{C}$ to obtain the elementary edges. It allows us to test all the constraints defining the domain and guarantees a better boundary coverage, but increases the number of tests.

3 Boundary Coverage Criteria

3.1 Definition of the Criteria

Let $D \neq \varnothing$ be a domain and $TS \subset D$ be a set of tests. We say that TS satisfies *the* **AB** *(All-Boundaries) criterion on* D if all the boundary states of D belong to TS, i. e. $\mathrm{Fr}(D) \subset TS$. We say that TS satisfies *the* **OB** *(One-Boundary) criterion on* D if at least one boundary state of D belongs to TS, i. e. $\mathrm{Fr}(D) \cap TS \neq \varnothing$.

We say that TS satisfies *the* **MD** *(Multi-Dimensional) criterion on* D if every variable $x_1, \ldots, x_n$ takes its minimum and its maximum on D in some states of TS.

For the next two criteria, we assume (1) and (5) as above. We say that TS satisfies *the* **AE** *(All-Edges) criterion on* D if every non-empty edge E_j has at least one point in TS, i. e. $E_j \cap TS \neq \varnothing$. This is equivalent to saying that TS satisfies the **OB** criterion on every non-empty edge E_j.

We say that TS satisfies *the* **AEMD** *(All-Edges Multi-Dimensional) criterion on* D if for every non-empty edge E_j, every variable $x_1, \ldots, x_n$ realizes its minimum and its maximum on E_j in some states of TS. This is equivalent to saying that TS satisfies the **MD** criterion on every non-empty edge E_j.

To give an example, we will consider the following test sets for the domain D defined by (4):

$$\begin{gathered} TS_1 = \{(-3,0),(-2,0),(-1,0),(0,0),(2,0),(3,0)\}, \\ TS_2 = \{(-2,4),(-3,4),(-4,3)\}, \\ TS_3 = \{(-3,4),(-2,0),(3,2)\}, \\ TS_4 = \{(-5,0),(5,0),(0,5)\}, \\ TS_5 = \mathrm{Fr}(D). \end{gathered} \tag{8}$$

The following table shows which criteria are satisfied for each TS_i:

	OB	**AE**	**MD**	**AEMD**	**AB**
TS_1	$\surd$	$\times$	$\times$	$\times$	$\times$
TS_2	$\surd$	$\times$	$\times$	$\times$	$\times$
TS_3	$\surd$	$\surd$	$\times$	$\times$	$\times$
TS_4	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
TS_5	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

3.2 Criteria Hierarchy

We will now compare the criteria. We say that a criterion A subsumes another criterion B, and we write $A \Rightarrow B$, if for any domain D, every test set $TS \subset D$ satisfying A, satisfies B. It is clear that the subsumption is transitive. One can easily find examples to show that the criteria **AE** and **MD** do not subsume one another.

Proposition 2 *We have*

$$\mathbf{AB} \Rightarrow \mathbf{AEMD},\ \mathbf{AEMD} \Rightarrow \mathbf{AE},$$
$$\mathbf{AEMD} \Rightarrow \mathbf{MD},\ \mathbf{AE} \Rightarrow \mathbf{OB},\ \mathbf{MD} \Rightarrow \mathbf{OB}.$$

These subsumptions are strict. The criteria subsumption lattice is given by the following diagram (an edge between two criteria means that the higher one strictly subsumes the lower one):

$$\begin{tikzcd} & \mathbf{AB} \arrow[d, dash] & \\ & \mathbf{AEMD} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathbf{AE} \arrow[dr, dash] & & \mathbf{MD} \arrow[dl, dash] \\ & \mathbf{OB} & \end{tikzcd} \tag{9}$$

Proof in Annexe (Section 10).

4 Test Selection Algorithms

4.1 Description of the Method

Let $D \subset \mathbb{Z}^n$ be a discrete domain defined as in (1) and (5). Although $\mathrm{Fr}(D)$ is often easy to determine in simple examples, boundary state generation in the general case can be a difficult problem. To decide if a given point $A \in D$ is a boundary state of D, one should check if all the points of $V(A) \backslash \{A\}$ belong to D. If D is defined by $\mathcal{C}$ as in (5), one should make l verifications of the constraints $\mathcal{C}_j$ for each of the $2n$ points $B \in V(A) \backslash \{A\}$. This is too expensive to be practical.

We propose here an efficient method of boundary state generation based on the cost function minimization. This is implemented in the BZ-TT environment. We choose a cost function $f(x_1, \ldots, x_n)$ and minimize (or maximize) f on the domain D. As the maximization of f is equivalent to the minimization of the function $(-f)$, we will talk about the minimization only. For an appropriate function f, the minimization of f makes it possible to find a boundary state $A \in D$ such that $f(A)$ is the minimal value of f on D, i. e.

$$f(A) = min\{\, f(Y) \mid Y \in D\,\}.$$

CLP (constraint logic programming) techniques can be used to efficiently find the minimum solutions of the cost function.

The choice of the cost function f is certainly very important. We would like to choose a function f such that the minimization of f on every discrete domain D gives us boundary states of D. Of course, it will not be true for an arbitrary function.

4.2 How to Choose a Cost Function

Consider the domain $D = [-10, 5] \cap \mathbb{Z}$. This domain has two boundary states $x = -10$ and $x = 5$. The first condition which seems natural to impose on cost functions is the following: *a cost function must be defined on the domain.* For example, the function $f(x) = \sqrt{x}$ is not suitable because it is not defined everywhere on D. Another example of an inappropriate cost function for the same domain D is $f(x) = x^2$. The minimal value of f on D is $f(0) = 0$, but $x = 0$ is not a boundary state of D. In this case, the function f has a minimum in $D \setminus \mathrm{Fr}(D)$ and the minimization of f gives us this point inside the domain D instead of a boundary state. To avoid this problem, we will impose another condition on f: *a cost function must not have a minimum inside the domain.*

To ensure that our function works for any domain $D \subset \mathbb{Z}^n$, we require that these two conditions hold on $\mathbb{Z}^n$. The counter-examples above show that these two conditions are necessary, but actually they are also sufficient:

Proposition 3 *Let $f : \mathbb{Z}^n \to \mathbb{R}$ be a function. Suppose that f does not have any local minimum, i. e.*

$$\begin{gathered} \text{for every } X \in \mathbb{Z}^n, \\ f(X) \neq min\{\, f(Y) \mid Y \in V(X)\,\}. \end{gathered} \tag{10}$$

Then for every discrete domain D, the minimization of f on D gives a boundary state of D. Proof in Annexe (Section 10).

Corollary 4 *Let $f : \mathbb{Z}^n \to \mathbb{R}$ be a non-zero linear function defined by*

$$f(x_1, \ldots, x_n) = a_1 x_1 + \ldots + a_n x_n,\ a_i \in \mathbb{R}.$$

Then for every discrete domain D, the minimization (or maximization) of f on D gives a boundary state of D. Proof in Annexe (Section 10).

4.3 Practical Test Selection

The **AB** criterion is very strong and seems difficult to satisfy for real-size domains. It would mean, firstly, to find all the boundary states of D, and secondly, to obtain and execute a very big number of tests.

To satisfy the **OB** criterion, it is sufficient to find one boundary state of D by minimizing a cost function satisfying the assumptions of Proposition 3. We cannot in general say which cost function is the best one because it depends on the test goals and the given domain. While generating only one boundary state per domain, we prefer to take a function involving all the variables e. g.

$$f(x_1, \ldots, x_n) = x_1 + \ldots + x_n.$$

f is suitable by Corollary 4 and allows us to obtain vertices on edge intersections in the case where some variables are not linked by the constraints, like in the following example:

$$\begin{gathered} I = \mathbb{Z} \times \mathbb{Z},\ n = l = 2, \\ \mathcal{C}_1 : 0 \leqslant x_1 \leqslant 50, \\ \mathcal{C}_2 : -5 \leqslant x_2 \leqslant 10, \end{gathered} \tag{11}$$

where

$$f(0, -5) = min\{f(x_1, x_2) \mid (x_1, x_2) \in D\}$$

and $(0, -5)$ is a vertex belonging to both edges of D. For the domain defined by (4), this method would give the boundary state $(-5, 0)$ belonging to the first and the third edges.

AE is a reasonable compromise between the **AB** and **OB** criteria. It allows us to test every edge without taking all its points. Therefore, **AE** needs at most l tests for D. We developed algorithms for this criterion in some particular cases including that of linear constraints. For example, for a constraint

$$a_1 x_1 + \ldots + a_n x_n \geqslant b,$$

we minimize the function

$$f = a_1 x_1 + \ldots + a_n x_n$$

(suitable by Corollary 4) on D to find a boundary state on the corresponding edge. A good approximation of **AE** in many cases is given by **MD**. To realize the **MD** criterion, we minimize and maximize on D each of the cost functions

$$f_1 = x_1,\ f_2 = x_2,\ \ldots,\ f_n = x_n. \tag{12}$$

This is always possible by Proposition 1 and gives up to $2n$ boundary states (some of them can coincide as in the example of Section 2.1). **AEMD** is stronger than **AE**, it tests every edge of D in every dimension and requires in the worst case $2nl$ tests.

Some other criteria can be of interest when one wants to reduce the number of tests. For example, if the system is particularly unstable for small (respectively, big) values of variables, one can apply the **OSMD** *(One-Sided Multi-Dimensional) criterion* which is **MD** with the minimum condition only (respectively, with the maximum condition only). To implement this criterion, we will only minimize (respectively, maximize) the functions (12) and obtain n tests. Another possibility is to apply the **MD** criterion only on some variables. For example, one can take one variable in each group of independent variables (where there are several groups of variables that are linked by some constraints within a group, but the variables of different groups never appear in the same constraint). We call this the **AGMD** *(All-Groups Multi-Dimensional) criterion*. In (9), **OSMD** and **AGMD** will be situated between **OB** and **MD**.

4.4 Implementing Test Selection Procedures using Constraint Programming

The OB and MD criteria are implemented in the BZ-TT environment [19, 2], where they can be used by the test engineer to control the generation of tests from Z, B and UML/OCL models.

These test selection techniques use existing constraint solvers (Boolean and finite domains). In BZ-TT, due to the particular features of the set-oriented specification notations, we also developed our own solver to handle constraints over sets, relations and mappings. This solver, called CLPS [6], augments the capabilities of and co-operates with the integer finite domain solver of SICStus Prolog [27]. It makes it possible to efficiently implement and compute on discrete domains the boundary selection method presented in this paper.

5 Test Generation on the Triangle Example

In this section, we illustrate the test selection using boundary coverage criteria from the 8 effect predicates of the triangle example (Fig. 2, Section 2.1).

Fig. 7 shows the test cases that result from applying test selection using the boundary criteria **OB**$_{min}$ (**OB** using only minimization), **OB**$_{max}$ (**OB** using only maximization), and **MD**. Note that we set an arbitrary value to the constant $MAXSIZE$ ($MAXSIZE = 10$) in the B model in order to compute concrete boundary states on the input variables. For each predicate, the triples give the test inputs for the variables (s1,s2,s3) and the test Oracle gives the expected value for the output variable.

We can observe that the **OB** criterion makes it possible to test each effect predicate once at a minimum point (**OB**$_{min}$), or once at a maximum point (**OB**$_{max}$).

OB$_{min}$**:** $PS_1 : (1,1,2), Oracle : kind = invalid$
$PS_2 : (2,1,1), Oracle : kind = invalid$
$PS_3 : (1,2,1), Oracle : kind = invalid$
$PS_4 : (1,1,1), Oracle : kind = equilateral$
$PS_5 : (1,2,2), Oracle : kind = isosceles$
$PS_6 : (2,1,2), Oracle : kind = isosceles$
$PS_7 : (2,2,1), Oracle : kind = isosceles$
$PS_8 : (2,3,4), Oracle : kind = scalene$

OB$_{max}$**:** $PS_1 : (1,9,10), Oracle : kind = invalid$
$PS_2 : (10,1,9), Oracle : kind = invalid$
$PS_3 : (1,10,9), Oracle : kind = invalid$
$PS_4 : (10,10,10), Oracle : kind = equilateral$
$PS_5 : (9,10,10), Oracle : kind = isosceles$
$PS_6 : (10,9,10), Oracle : kind = isosceles$
$PS_7 : (10,10,9), Oracle : kind = isosceles$
$PS_8 : (8,9,10), Oracle : kind = scalene$

MD: $PS_1 : (1,1,2), (9,1,10), (1,9,10), (1,1,10),$
$Oracle : kind = invalid$
$PS_2 : (2,1,1), (10,9,1), (10,1,9), (10,1,1),$
$Oracle : kind = invalid$
$PS_3 : (1,2,1), (9,10,1), (1,10,9), (1,10,1),$
$Oracle : kind = invalid$
$PS_4 : (1,1,1), (10,10,10),$
$Oracle : kind = equilateral$
$PS_5 : (1,2,2), (10,6,6), (1,10,10),$
$Oracle : kind = isosceles$
$PS_6 : (2,1,2), (10,1,10), (6,10,6),$
$Oracle : kind = isosceles$
$PS_7 : (2,2,1), (10,10,1), (6,6,10),$
$Oracle : kind = isosceles$
$PS_8 : (2,3,4), (10,2,9), (3,2,4), (2,10,9),$
$(3,4,2), (2,9,10),$
$Oracle : kind = scalene$

Figure 7. OB$_{min}$, OB$_{max}$ and MD Test cases.

The **MD** criterion gives a maximum and a minimum for each input variable, but results in 29 test cases. One can check that in this example **MD** test sets satisfy the **AE** criterion also. Applying **AEMD** would give more test cases.

To complete the analysis of our triangle test set, we used the method of mutation testing (e.g. [23]) on the formal model itself. The mutations are defined by all possible confusions:

- between x, $x+1$ and x, $x-1$ for each variable x;
- between $+$, $-$ and $*$;
- between $<$, $>$, $=$, $\leqslant$, $\geqslant$ and $\neq$;
- between $\vee$ and $\wedge$.

In this way we obtained 89 mutant specifications. Each of them contains one *domain error* (see [30]). Then we tested the mutants using the test methods based on our boundary coverage criteria (**OB**$_{min}$, **OB**$_{max}$ and **MD**), and using the following test suite that guarantees to cover all the effect predicates but without boundary evaluation (**WB**).

WB: $PS_1 : (1, 3, 6), Oracle : kind = invalid$
$PS_2 : (4, 1, 2), Oracle : kind = invalid$
$PS_3 : (2, 8, 5), Oracle : kind = invalid$
$PS_4 : (9, 9, 9), Oracle : kind = equilateral$
$PS_5 : (4, 3, 3), Oracle : kind = isosceles$
$PS_6 : (5, 8, 5), Oracle : kind = isosceles$
$PS_7 : (8, 8, 6), Oracle : kind = isosceles$
$PS_8 : (7, 5, 4), Oracle : kind = scalene$

Figure 8. WB Test cases.

The following table shows the fault-detection effectiveness for all these criteria:

Test method	**WB**	**OB**$_{max}$	**OB**$_{min}$	**MD**
Total faults	89	89	89	89
Detected faults	61	69	81	89
% of detected faults	69%	78%	91%	100%

These comparison results show the effectiveness of our criteria on the triangle example. **MD** appear to be the most efficient criterion: all the mutant specifications are killed using this criterion on this example.

These results are representative of the results obtained on several industrial applications: the **MD** criterion generates more test cases than **WB**, **OB**$_{max}$ or **OB**$_{min}$, but gives in the end better fault-detection results. In another way, with the same number of test cases, more defaults are detected using boundary criteria.

We can note that the **OB** criterion gives different results using maximization or minimization. Such a difference between minimization and maximization (already noticed on several industrial applications) is directly due to the specification, and cannot be used to conclude that minimization (resp. maximization) is more efficient than maximization (resp. minimization).

6 Related Work

Boundary testing is widely used as an informal heuristic during manual test design, but has not previously been formalized as coverage criteria.

However, boundary testing is similar to the fault model of domain testing [30], [3, Chapter 7]. These approaches use test heuristic strategies like "test the extreme points" or "test the extreme point combinations". Our work systematizes these approaches by giving formalized criteria and methods to compute it.

In code-based domain testing, so-called ON-OFF-points are often used in *domain testing* for continuous domains [30]. While testing an implementation with discrete variables, one cannot execute a non-integer test case. Therefore, taking an OFF-point a small distance ε from the boundary is of no interest, because in general the obtained test case is not integer and cannot be executed. Generating of ON-points (boundary states) in this case is also more complex. Consider for example a discrete domain $D \subset \mathbb{Z}^2$ defined by

$$x_1^2 + x_2^2 \leqslant 11.$$

Contrary to the continuous case, it is not sufficient to solve the corresponding equation. Indeed, the equation

$$x_1^2 + x_2^2 = 11$$

has no solutions in $\mathbb{Z}^2$ and hence does not describe the boundary of D.

Due to the BZ-TT partitioning method, we do not need to separately consider OFF-points, i.e. points just outside the domain D of an effect predicate. Indeed, if a point A just outside D satisfies the invariant $\mathcal{I}$, it is a boundary state of the domain D' of some other effect predicate. Such points are automatically covered since all effect predicates are used to generate tests. If A does not satisfy $\mathcal{I}$, we cannot compute, according to the specification, an expected behaviour to assign a test verdict. Therefore, no test case is generated for A.

A partially automated boundary test generation system is described in [15]. 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. An important difference from our approach is that they 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, our search for each boundary test case considers the precondition, which means that we always generate valid test inputs, so obtain much more precise coverage of the real (semantic) boundary points. [15] also defines a family of *perimeter* strategies (*k-per*), where *1-per* holds one variable at a boundary value but allows the others to vary. This has some similarity to our multi-dimensional strategy. Another important difference from our approach is that the input space in [15] is supposed to be a product of intervals. We emphasize that no special restriction on the form of the discrete domain D is needed in our results.

Few Model-Based Test Generators offer boundary-value test selection strategies. Currently, we know only of our BZ-TT environment and the T-Vec tool [28] which implement automated boundary-value testing from formal models.

7 Discussion

Why do we need boundary-oriented model-based testing?
Covering all effect predicates of a formal model is very similar to cover all independent paths in a program. In code-based testing, path coverage is unfeasible, due to combinatorial explosion. But, in model-based testing, because there are no loops nor sequential composition operators, it is realistic (and necessary) to achieve model path coverage. Our experience with half of dozen of real-size industrial applications is that the number of effect predicates was from one hundred to 3 thousand. So, covering all the effect predicates of such model partitioning is possible. Each effect predicate defines an equivalent behavioral class of the model on the basis of partitioning of each modelled operation. The question is then which data values we have to select invoking the effect predicate?[1]. We can either invoke an arbitrary consistent nominal value for each state or input variable. Or we can choose an extremum value for these variables domains. This is boundary model-based testing.

Do boundary values help to find more errors in the system under test?
We demonstrated several times during industrial case-studies that, for the same number of test cases, to exercising the min or max values of domain variables helps to detect more faults. The reason is that although arbitrary values and boundary values both exercise the system behavior related to the effect predicate, boundary-values detect more faults on the conditions of the decisions in the system under test control flow. Actually, it is a well known testing heuristic in functional testing, and every test practitioner book recommends it.

Do we need coverage criteria for boundary-oriented model-based testing?
The coverage criteria can be used to measure the adequacy of an existing test set. But more importantly, the coverage criteria can be used to control the test generation process. The test engineer can choose a particular criterion that seems appropriate for the whole system under test or for a particular postcondition or predicate within its specification, and the appropriate number of tests will be generated to satisfy that criterion. This gives a clear and understandable way of controlling model-based test generation.

[1]Note that not all effect predicates are feasible from the initial state, so we compute a preamble to put the model in a state where the effect predicate could be invoked. See [11] for preamble computation in the BZ-TT test generator □

8 Conclusions and Future Work

We have introduced a hierarchy of data-oriented boundary coverage criteria, and given test generation algorithms for them based on ordering functions. The main three contributions of this paper are:

1. formalizing the common heuristic of boundary testing to obtain a hierarchy of boundary coverage criteria. The coverage criteria range from OB (a single boundary point) to AB (all boundary points). The formalization is general, and does not require linear constraints or independent constraints.

2. defining several intermediate boundary coverage criteria based on the set of variables that appear in the specification (MD) and the set of constraints that appear in the specification (AE), plus combinations of those (AEMD). These intermediate criteria are important in practice, because engineers often want more than one test per effect (OB), but AB produces too many tests to be practical on complex applications. The intermediate criteria give systematic guidelines for producing a linear number of tests. Our case-studies and industry experience have demonstrated that these criteria are useful and that they detect more errors than OB.

3. describing practical techniques for generating test sets from a given boundary coverage criterion, by minimizing and maximizing cost functions. This is important, because it makes test generation more efficient, since the given criterion can be used to guide the generation of tests, and unnecessary tests are not generated.

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 specification notations. This boundary coverage criteria scheme has been mostly implemented in the BZ-TT environment [8] using constraint logic programming techniques. It generates boundary states covering each part of each postcondition, then finds tests that satisfy those boundary goals. BZ-TT also addresses the sequencing problem: computing a sequence of operations from the initial state to the boundary state [11]. This enables it to produce tests that activate each postcondition when the system is in a boundary state. The BZ-TT environment has been used for half of dozen of industrial applications between 1999 and 2003 in the area of smart cards software (some results are shown in [4]), automotive embedded systems and bank electronic payment. All these real size formal models involve large combinatorial state space. The boundary coverage criteria also help the validation engineer to efficiently drive the automated generation process.

The BZ-TT technology is now mature enough to be used in industrial setting for animation and model-based testing. So, this technology is currently being transferred to a start-up, LEIRIOS Technologies [21], which is industrializing the tool, marketing it and using it for outsourced testing. This industrial tool, called Leirios Test Generator, applies various model coverage criteria schemes including these boundary criteria.

9 Acknowledgement

This work is partially supported by Schlumberger/Axalto Smart Card R&D and PSA Peugeot Citroen.

10 Annexe

In this section we give proofs for our results.

Proof of Proposition 1 : For convenience of notation, we assume that $i = 1$. We will prove that there exists a boundary state $A = (a_1, \dots, a_n)$ of a discrete domain D such that

$$a_1 = min\{\, x_1 \mid (x_1, \dots, x_n) \in D \,\}.$$

The proof for the maximal value can be done in the same manner.

Let $L = \{\, x_1 \mid (x_1, \dots, x_n) \in D$ for some $x_2, \dots, x_n\} \subset \mathbb{Z}$ be the projection of D on x_1. As D is bounded, L is finite, hence there exists a minimal value $a_1 = min\, L$. By the definition of L, there exists a point $A = (a_1, \dots, a_n) \in D$. On the other hand,

$$(a_1 - 1, a_2, \dots, a_n) \in V(A),$$
$$(a_1 - 1, a_2, \dots, a_n) \notin D,$$

so A is a boundary state of D. $\square$

Proof of (6): Let us first prove that $E_j \subset \mathrm{Fr}(D)$. Let $A \in E_j = \mathrm{Fr}(D_j) \cap D$. Then there exists $B \in V(A)$ such that $B \notin D_j$. As $D \subset D_j$, we have $B \notin D$. It follows that $A \in \mathrm{Fr}(D)$. We proved that $E_j \subset \mathrm{Fr}(D)$.

We will now prove that

$$\mathrm{Fr}(D) \subset \bigcup_{j=1}^{l} E_j.$$

Let $A \in \mathrm{Fr}(D)$. Then $A \in D$ and there exists $B \in V(A)$ such that $B \notin D$. As $D = \bigcap_{j=1}^{l} D_j$, we have $B \notin D_k$ for some $k \in \{1, 2, \dots, l\}$. It follows that $A \in \mathrm{Fr}(D_k) \cap D = E_k$. We proved that $\mathrm{Fr}(D) = \bigcup_{j=1}^{l} E_j$. $\square$

Proof of Proposition 2: Let $D \neq \varnothing$ be a discrete domain and TS be a test set. As before, we assume (5) (while speaking about the $\mathbf{AE}$ and $\mathbf{AEMD}$ criteria). First we will prove that $\mathbf{AB} \Rightarrow \mathbf{AEMD}$. Let E_j be a non-empty edge of D. By (6), $E_j \subset \mathrm{Fr}(D)$. If TS satisfies $\mathbf{AB}$, then $\mathrm{Fr}(D) \subset TS$, hence, $E_j \subset TS$. We see that TS satisfies $\mathbf{AEMD}$.

To show that $\mathbf{AEMD} \Rightarrow \mathbf{AE}$, we suppose that TS satisfies $\mathbf{AEMD}$. Then every non-empty edge E_j has at least one point in TS, for example, that minimizing the first variable.

Let us now show that $\mathbf{AEMD} \Rightarrow \mathbf{MD}$. If TS satisfies $\mathbf{AEMD}$, then for every non-empty edge E_j every variable realizes its maximum and its minimum on E_j in some points of TS. Let x_i be one of the variables. The minimal value of x_i on $\mathrm{Fr}(D)$ is exactly its minimal value on some edge E_j, because the edges compose the boundary $\mathrm{Fr}(D)$ by (6). We see that every variable realizes its minimum on $\mathrm{Fr}(D)$ in some points of TS. We prove in the same manner that every variable realizes its maximum on $\mathrm{Fr}(D)$ in some points of TS, therefore TS satisfies $\mathbf{MD}$.

The subsumption $\mathbf{AE} \Rightarrow \mathbf{OB}$ is trivial because if TS satisfies $\mathbf{AE}$, we can find at least one boundary state in TS (for example, that corresponding to the first non-empty edge). The subsumption $\mathbf{MD} \Rightarrow \mathbf{OB}$ is also trivial because if TS satisfies $\mathbf{MD}$, TS contains, for example, the boundary state minimizing the first variable.

The test sets (8) for the domain D defined by (4) show that the subsumptions $\mathbf{AB} \Rightarrow \mathbf{AEMD}$, $\mathbf{AEMD} \Rightarrow \mathbf{AE}$, $\mathbf{AE} \Rightarrow \mathbf{OB}$, $\mathbf{MD} \Rightarrow \mathbf{OB}$ are strict. The reader may find examples to prove that $\mathbf{AEMD} \Rightarrow \mathbf{MD}$ is also strict. (Such examples do not exist for the domain D of (4).) $\square$

Proof of Proposition 3 : Let $D \subset \mathbb{Z}^n$ be a discrete domain and $A \in D$ be a minimum of f on D, i. e.

$$f(A) = min\{\, f(Y) \mid Y \in D \,\}.$$

To obtain a contradiction, suppose that A is not a boundary state of D. Then $V(A)$ has no points in $\mathbb{Z}^n \backslash D$, therefore $V(A) \subset D$. As $f(A)$ is the minimal value of f on D, $f(A)$ is also the minimal value of f on $V(A)$:

$$f(A) = min\{\, f(Y) \mid Y \in V(A) \,\},$$

contrary to (10). $\square$

Proof of Corollary 4 : The corollary is a consequence of Proposition 3 as the linear function f does not have any local minimum in $\mathbb{Z}^n$. As the function

$$-f = -a_1 x_1 - \dots - a_n x_n$$

is also linear and the maximization of f is equivalent to the minimization of $(-f)$, the maximization of f on D gives a boundary state of D as well. $\square$

References

[1] J.-R. Abrial. *The B-BOOK: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0 521 49619 5.

[2] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.

[3] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[4] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11.11 standard case-study. *The Journal of Software Practice and Experience*, 2004. Accepted for publication.

[5] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation: the Java Card transaction mechanism case study. In *Proceedings of the International Conference on Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003. Springer Verlag.

[6] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B – A constraint solver for B. In *Proceedings of the ETAPS'02 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 188–204, Grenoble, France, April 2002. Springer Verlag.

[7] F. Bouquet, B. Legeard, N. Vacelet, and M. Utting. Faster Analysis of Formal Specifications. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, LNCS, page to appear, Seattle, USA, November 2004. Springer Verlag.

[8] The BZ-TT web site. `http://lifc.univ-fcomte.fr/~bztt`, September 2003.

[9] N. Caritey, L. Gaspari, B. Legeard, and F. Peureux. Specification-based testing – Application on algorithms of Metro and RER tickets (confidential). Technical Report TR-03/01, LIFC - University of Franche-Comté and Schlumberger Besançon, 2001.

[10] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[11] S. Colin, B. Legeard, and F. Peureux. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):to appear, 2004.

[12] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[13] A. Gargantini and E. Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *JUCS*, 10(8), November 2001.

[14] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *The Journal of ACM Transaction on Software Engineering and Methodologies*, 5(4):293–333, October 1996.

[15] D. Hoffman, P. Strooper, and L. White. Boundary values and automated component testing. *The Journal of Software Testing, Verification and Review*, 9(1):3–26, 1999.

[16] H. Hong, Y. Kim, S. Cha, D. Bae, and H. Ural. A test sequence selection method for statecharts. *The Journal of Software Testing, Verification and Reliability*, 10(4):203–227, December 2000.

[17] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2^{nd} edition, 1990.

[18] B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[19] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark, July 2002. Springer Verlag.

[20] B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *The Journal of Software Testing, Verification and Reliability*, 14(2):81 – 103, 2004.

[21] The Leirios web site. `http://www.leirios.com`, September 2003.

[22] G. Myers. *The Art of Software Testing*. Wiley-InterScience, 1979.

[23] A. Offutt and S. Lee. An Empirical Evaluation of Weak Mutation. *The Journal of IEEE Transaction on Software Engineering*, 20(5):337–344, May 1994.

[24] A. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, addison-wesley edition, 1999.

[26] J. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, 2^{nd} edition, 1992. ISBN 0 13 978529 9.

[27] Swedish Institute of Computer Sciences. *SICStus Prolog 3.8.7 manual documents*, October 2001. `http://www.sics.se/sicstus.html`.

[28] T-Vec Tester. `http://www.t-vec.com/`, 2003.

[29] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[30] L. White and E. Cohen. A Domain Strategy for Computer Program Testing. *The journal IEEE Transactions on Software Engineering*, 6:247–257, 1980.

[31] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.