Coverage Criteria for Set-Based Specifications

Maximiliano Cristiá [1]
Joaquín Cuenca [2]
Claudia Frydman [3]

Data de submissão: 20.12.2014
Data de aceitação: 17.08.2015

Abstract: Model-based testing (MBT) studies how test cases are generated from a model of the system under test (SUT). Many MBT methods rely on building an automaton from the model and then they generate test cases by covering the automaton with different path coverage criteria. However, if a model of the SUT is a logical formula over some complex mathematical theories (such as set theory) it may be more natural or intuitive to apply coverage criteria directly over the formula. On the other hand, domain partition, i.e. the partition of the input domain of model operations, is one of the main techniques in MBT. Partitioning is conducted by applying different rules or heuristics. Engineers may find it difficult to decide what, where and how these rules should be applied. In this paper we propose a set of coverage criteria based on domain partition for set-based specifications. We call them *testing strategies*. Testing strategies play a similar role to path- or data-based coverage criteria in structural testing. Furthermore, we show a partial order of testing strategies as is done in structural testing. We also describe an implementation of testing strategies for the Test Template Framework, which is a MBT method for the Z notation; and a scripting language that allows users to implement testing strategies.

[1] CIFASIS and UNR, Rosario, Argentina
`cristia@cifasis-conicet.gov.ar`
[2] UNR, Rosario Argentina
`cjoacuenca@gmail.com`
[3] LSIS-CIFASIS, Marseille, France
`claudia.frydman@lsis.org`

1 Introduction

Testing is the predominant verification technique used in the software industry. At the same time, testing is a time-consuming activity that needs many resources to produce good results. Since many years ago the testing community works on the automation of the testing process as a means to reduce its costs and improve its efficiency. Model-based testing (MBT) is a collection of testing methods that aim at generate test cases from the analysis of a model of the system under test (SUT) [31]. MBT methods have achieved impressive theoretical and practical results in recent years such as [20, 15, 30, 24, 32], to name just a few.

Some MBT methods generate test cases by first building an automaton from the model of the SUT and then covering the automaton with different criteria. For example, ProTest [26] is a test case generation tool based on B machines [1]. It first writes each given machine operation into DNF and then it builds a finite state machine (FSM) "whose initial node is the initial state of the B machine. Each node in the FSM represents a possible machine state and each edge is labeled by an operation". Finally, ProTest "traverses the FSM to generate a set of operation sequences such that each operation in the FSM appears in the generated sequences at least once". As another example consider the method proposed by Hierons et al. [19, 18]. In this case a Z specification [27] is accompanied by a Statechart [17] that represents all the possible execution paths of the operations defined in the Z specification. Then, some test sequence generation methods, based on FSM test techniques, are defined. These criteria are based on covering the paths of the Statechart in different ways. Since the Statechart is built from the Z specification then the test sequences will cover the Z specification. There are other MBT methods that generate some kind of automaton from specifications described in rich mathematical languages [13, 5].

As can be seen, even when the model of the SUT is (essentially) a logical formula based on a complex mathematical theory (i.e. a B machine or a Z operation) test cases are generated by covering the paths of an automaton derived from the model, and not by covering the structure of the formula. In a sense, there is an assumption that the logical formula is covered by covering the automaton generated from it. In this paper we would like to propose an alternative method that generates test cases by applying *criteria that cover directly the specification when it is given as a state-based model written in first-order logic over a set theory*—e.g. Z [27], B [1], VDM [23], Alloy [22] and others[4]. We call these coverage criteria *testing strategies*. Testing strategies are defined by conveniently assembling together rules for *domain partition*. Domain partition is a technique that divides the input space of the model operations into subsets from which test cases are generated [16, 29, 3, 21]. Our criteria do not need an automaton because they analyze the structure, semantics and types of the specification itself. Testing strategies can be organized in a partial order to help engineers

[4] For now on, we will refer to these languages as *set-based languages* and to their specifications as *set-based specifications*.

to select the most appropriate for each project. Furthermore, given that it may not be easy to apply rules for domain partition, the criteria presented in this paper relief engineers from understanding the internals of domain partition.

Besides, the paper also shows how testing strategies have been implemented in FASTEST [8], a tool that supports the Test Template Framework (TTF) [29] which, in turn, is a MBT method for the Z notation. The implementation of these coverage criteria includes the definition of a scripting language with which engineers can implement new testing strategies.

The paper starts by introducing the concept of recursive domain partition in Section 2. Recursive domain partition shows how partitioning rules can be applied together, so Section 3 describes some known partitioning rues. Testing strategies are introduced in Section 4 where they are accommodated in a partial order. Section 5 presents a prototype implementation of testing strategies in FASTEST, a MBT tool based on Z specifications. We discuss the contribution of this paper in Section 6. Our conclusions are presented in Section 7.

2 Recursive Domain Partition

In this section we show how test cases can be systematically generated from a set-based specification by applying *recursive domain partition*. This technique results from the combination of some rules for domain partition that are already available. Recursive domain partition is the basis on which coverage criteria for set-based specifications are defined.

Consider an operation that eliminates a symbol from a symbol table that associates symbols with integer numbers. A possible formal specification of this operation is the following logical formula:

$$(s? \in \operatorname{dom} st \land st' = \{s?\} \lhd st) \lor (s? \notin \operatorname{dom} st \land st' = st) \tag{1}$$

where $st : SYM \nrightarrow \mathbb{Z}$ is a partial function[5]; SYM is a given or uninterpreted sort; $s? : SYM$ is an input variable; st is a state variable; st' represents the value of st in the next state; and $\lhd$ is a relational operator called domain anti-restriction[6]. Therefore (1) can be interpreted as follows: symbol $s?$ is eliminated from symbol table st which associates elements of SYM with integer numbers. In other words, (1) describes a state transition of some state machine.

If formula (1) is the specification of some implementation, then the MBT theory says that engineers should analyze (1), instead of its implementation, to generate test cases to test the implementation. Note that it would be difficult to generate test cases from an automaton built from (1), as suggested by some MBT methods [26, 19, 18], because we have only one

[5] In this paper, partial functions are sets of ordered pairs as defined for instance in the Z and B notations.

[6] Formally, if $R : X \leftrightarrow Y$ is a binary relation between X and Y and $A : \mathbb{P}\,X$ is a subset of X, then $A \lhd R = \{(x, y) \in R \mid x \notin A\}$.

transition. On the other hand, the implementation of (1) is not trivial because, for instance, it will seldom be based on sets and set operators (such as $\lhd$). Sets and set operators will probably be implemented as arrays or linked lists and operations over them. Hence, the implementation of (1) is worth to be tested.

Following the foundation of structural testing we may ask to ourselves, how (1) can be *covered*? That is, how can we be sure that all aspects of (1) are going to be considered for test case generation? An answer given by some MBT methods is to use domain partition [29, 3]. First, the input domain of the specification is defined, then it is recursively partitioned into subsets called *test conditions* and finally one element of each test condition is taken as a test case[7]. For example, the input space of (1) is defined as the set:

$$IS = \{st : SYM \pfun \mathbb{Z};\ s? : SYM\} \tag{2}$$

that is, the set of all possible input values for the operation. We can partition it by taking the precondition of each disjunct in (1):

$$IS_1 = \{IS \mid s? \in \operatorname{dom} st\} \tag{3}$$

$$IS_2 = \{IS \mid s? \notin \operatorname{dom} st\} \tag{4}$$

From this partition we can generate, for example, the following two test cases:

$$TC_1 = \{IS_1 \mid st = \{(s_1, 3)\} \land s? = s_1\} \tag{5}$$

$$TC_2 = \{IS_2 \mid st = \emptyset \land s? = s_1\} \tag{6}$$

Note that test cases are bindings between variables and values—for example, $\{(s_1, 3)\}$ is bound to *st* in TC_1. Indeed, these values are such that *satisfy* the predicate of the corresponding test condition. This can be seen, for instance, by *expanding* IS_1 in TC_1:

$$TC_1 = \{IS \mid s? \in \operatorname{dom} st \land st = \{(s_1, 3)\} \land s? = s_1\} \tag{7}$$

However, are these test cases enough? Will they test the implementation of (1) thoroughly? Do they cover all the specification? Probably not because, for example, there is no test case removing an ordered pair from *st* when it has more than one element. So, if *st* is implemented as, say, a linked list TC_1 and TC_2 will not test whether the iteration over the list is correctly implemented or not. Is there something in (1) that indicates to us that this (and possibly others) test case is missing? Yes, it is. The iteration over the (possible) list implementing *st* is specified by both the $\lhd$ and dom operators, and we have not used them to generate test cases.

[7] Test conditions are also called test templates, test specifications, test classes, etc.

$$
\begin{array}{l}
R = \emptyset \\
R \neq \emptyset \land S = \emptyset \\
R \neq \emptyset \land S = \operatorname{dom} R \\
R \neq \emptyset \land S \neq \emptyset \land S \subset \operatorname{dom} R \\
R \neq \emptyset \land S \neq \emptyset \land S \cap \operatorname{dom} R = \emptyset \\
R \neq \emptyset \land S \neq \emptyset \land S \cap \operatorname{dom} R \neq \emptyset \land \operatorname{dom} R \subset S \\
R \neq \emptyset \land S \neq \emptyset \land S \cap \operatorname{dom} R \neq \emptyset \land \neg\ (\operatorname{dom} R \subseteq S) \land \neg\ (S \subseteq \operatorname{dom} R)
\end{array}
$$

Figure 1. Standard partition for $S \lhd R$

In this example we use $\lhd$ to guide the partitioning process. In order to do that we can define a so-called *standard partition* for $\lhd$ as shown in Figure 1. Standard partitioning (SP) is another rule for domain partition. It attaches a standard partition to each mathematical operator that is seldom provided by programming languages. The standard partition is defined by giving conditions on the operands of the operator. Then the standard partition is applied to a particular expression in the specification where the operator appears. For example, we can apply the standard partition of $\lhd$ of Figure 1 to the expression $\{s?\} \lhd st$ of formula (1). Applying the standard partition means to substitute the names of the operands in the standard partition by the operands in the expression and building the corresponding subsets. For example, we substitute R by st and S by $\{s?\}$ and use this to partition IS_1 as follows:

$$IS_1^1 = \{IS_1 \mid st = \emptyset\} \tag{8}$$

$$IS_1^2 = \{IS_1 \mid st \neq \emptyset \land \{s?\} = \emptyset\} \tag{9}$$

$$IS_1^3 = \{IS_1 \mid st \neq \emptyset \land \{s?\} = \operatorname{dom} st\} \tag{10}$$

$$IS_1^4 = \{IS_1 \mid st \neq \emptyset \land \{s?\} \neq \emptyset \land \{s?\} \subset \operatorname{dom} st\} \tag{11}$$

$$IS_1^5 = \{IS_1 \mid st \neq \emptyset \land \{s?\} \neq \emptyset \land \{s?\} \cap \operatorname{dom} st = \emptyset\} \tag{12}$$

$$IS_1^6 = \{IS_1 \mid st \neq \emptyset \land \{s?\} \cap \operatorname{dom} st \neq \emptyset \land \operatorname{dom} st \subset \{s?\}\} \tag{13}$$

$$\begin{aligned} IS_1^7 = \{IS_1 \mid {} & st \neq \emptyset \land \{s?\} \cap \operatorname{dom} st \neq \emptyset \land \neg\ \operatorname{dom} st \subseteq \{s?\} \\ & \land \neg\ \{s?\} \subseteq \operatorname{dom} st\} \end{aligned} \tag{14}$$

This is an example of recursive domain partition. That is, the input space of an operation is partitioned with some partitioning rule, then one or more of the subsets thus obtained is partitioned by another partitioning rule, and so forth. Note that test conditions generated by recursive domain partition include all the previous conditions. For example, if we expand IS_1 inside IS_1^4 we get:

$$IS_1^4 = \{IS \mid s? \in \operatorname{dom} st \land st \neq \emptyset \land \{s?\} \neq \emptyset \land \{s?\} \subset \operatorname{dom} st\} \tag{15}$$

where the first conjunct corresponds to IS_1. In other words, each partition rule adds more conditions.

In particular, observe that now a test case derived from IS_1^4 will test whether removing an ordered pair from a symbol table containing more than one element is correct or not, which is the missing test case analyzed above. Indeed, the predicate of IS_1^4 can only be satisfied if *st* contains more than one ordered pair because it demands $\{s?\}$ to be a proper subset of the domain of *st*. So, for example, the following test case belongs to IS_1^4:

$$TC_3 = \{IS_1^4 \mid st = \{(s_1, 3), (s_2, 4)\} \land s? = s_1\} \tag{16}$$

Another observation that it is worth to be noticed regards the fact that some test conditions generated by recursive domain partition are unsatisfiable. For example, IS_1^2 and IS_1^5 are some of the unsatisfiable test conditions generated after applying SP. Since it is impossible to get a test case from an unsatisfiable test condition, they must be eliminated before applying another partitioning rule. The automatic elimination of unsatisfiable test conditions as well as the generation of test cases from (satisfiable) test conditions in the context of set-based specifications and recursive domain partition have been extensively studied elsewhere [8, 11].

Surely more test cases are needed to test the implementation of (1) but we think we have make it clear that mathematical specifications contain enough information as to derive a good test case set without necessarily resorting to an automaton. Furthermore, this technique can be applied unit by unit[8], and not necessarily to the whole interface of a component or system.

3 Partitioning Rules

In this section we briefly describe the partitioning rules mentioned above as well as other rules that are available for recursive domain partition. The set of partitioning rules depends on the syntax and semantics of the set-based language being used. The rules shown here can be applied to languages such as Z and B. There are more partitioning rules for such notations but for the sake of brevity we do not mention them here [9]. Users of other specification languages may adapt these rules and they may define others tailored to their specific languages.

Recall that in this context, a partitioning rule is a systematic way of deriving test conditions. A partitioning rule is always applied over a set of inputs which is subdivided in two or more subsets by the rule. Each of these subsets is characterized by a predicate, called *characteristic predicate*.

[8] In the sense of unit testing.

The test conditions named IS_1 and IS_2 in the previous section were obtained by a trivial application of a widely known partitioning rule called *Disjunctive Normal Form* (DNF). DNF writes the operation in DNF and then generates a test condition for each disjunct in the DNF. The characteristic predicate of each of these test conditions is the precondition of the corresponding disjunct in the DNF. The rationale behind the DNF tactic is to generate a test condition for each major logical alternative specified for an operation. For example, as was shown in the example developed above, DNF generates a test condition for each important precondition making it possible to test all the successful and error conditions.

However, a set-based specification is not only a logical formula but it also makes heavy use of many complex mathematical operators, particularly those of the set theory. Usually, the implementation of these operators is far from trivial. For example, developers prefer to use lists as if they were finite sets rather than using a set data structure that might be provided by the standard library of the programming language. In doing so, the implementation of some set operators tends to be non-trivial. DNF cannot generate test conditions to thoroughly exercise such implementations because it does not deal with mathematics but only with the logical structure of the specification [29]. This is the rationale behind the standard partition (SP) rule. By proposing a set of conditions with which the implementation of complex mathematical operators will be tested, the combination between DNF and SP provides a good coverage.

Enumeration Types (ET) helps to partition an input domain when variables whose type is enumerated are used. If x is a variable of an enumerated type, T, ET generates test conditions whose characteristic predicates are of the form $x = val$ for each $val \in T$. In this way, ET guarantees that the implementation will be exercised on all these values, which are usually part of conditional expressions and represent important states or operational conditions of the system.

Numeric Ranges (NR) waits for an arithmetic expression, *expr*, and an ordered list of numbers, $n_1, \dots n_k$, and generates the following partition: $expr < n_1$, $expr = n_1$, $n_1 < expr < n_2$, $\dots$, $expr = n_i$, $n_i < expr < n_{i+1}$, $expr = n_{i+1}$, $\dots$, $expr < n_k$, $expr = n_k$ and $n_k < expr$. NR is very useful, for instance, to test how programs behaves when numeric variables reach or go beyond their implementation limits.

Example 1 *In a C program a variable of type* `short` *can assume values in the range* $[-32768, 32767]$. *So, it would be reasonable to test the program with values less than, equal and greater than* -32768 *and* 32767 *for each input or state variable of type* `short`. *If one of such variables is* `x`, *likely, it was abstracted at the specification level as an expression of type* $\mathbb{Z}$. *For instance, a potential C implementation of variable st introduced in* (1) *might be as follows a singly-linked list whose nodes are as follows:*

```
struct st_node {char* sym; short val; struct st_node* nxt}
```

Therefore, if one wants to test the behavior of the program when a `sym` *has a* `val` *in the limits of the range for* `short`*, then the following list of values can be used* $[-32768, 32767]$ *to test the precondition* $s? \in \operatorname{dom} st$*. In this case NR would generate, for example,* $s? < -32768$ *as a test condition.*

In Set Extension (ISE) applies to specifications including preconditions of the form $expr \in \{expr_1, \dots, expr_n\}$. In this case, it generates $n + 1$ test conditions such that $expr = expr_i$, for i in $1 \mathrel{..} n$, and $expr \notin \{expr_1, \dots, expr_n\}$ are their characteristic predicates.

4 A Partial Order of Testing Strategies

Although domain partition is not a very complex activity, engineers need to analyze the specification, to select some partitioning rules and to decide what expressions, operators, variables, etc. are going to be used when these rules are applied. Besides, engineers working with domain partition do not have criteria that tell them "how much" the SUT is going to be tested. So far, they only have a set of partitioning rules only related through recursive domain partition. Our proposal in this regard is to define so-called *testing strategies*. A testing strategy uses one or more partitioning rules in such a way that some significant part of the specification is covered. In this sense, testing strategies are specification-based covering criteria.

As with coverage criteria of structural testing, each testing strategy aims at covering some aspect of a set-based specification. Following with the analogy, some testing strategies cover aspects already covered by others but add more aspects to be covered. For instance, in structural testing the criterion known as condition coverage covers all the edges of the control flow graph of the program and all the truth values of the conditions appearing in conditional sentences [14]. This means that condition coverage includes edge coverage but adds some more coverage. In set-based specifications there are three general aspects to be covered: a) the logical structure of the specification; b) the mathematical operators; and c) the type system.

The fact that some structural testing coverage criteria include others has been formalized as a partial order. In effect, Rapps and Weyuker [25] arrange coverage criteria according to a partial order indicating when a given criterion, in general, covers more than others. Using the work of as Rapps and Weyuker as an inspiration, we organized testing strategies according to a partial order as is depicted in Figure 2. The strategies closer to the bottom of the graph are those that produce a better coverage and those closer to the root produce a worse coverage. Informally, the strategies are as follows:

- BASIC-FUNCTIONS applies only disjunctive normal form (DNF) so it covers the logical structure of the specification. Since all the other strategies are stronger than this one

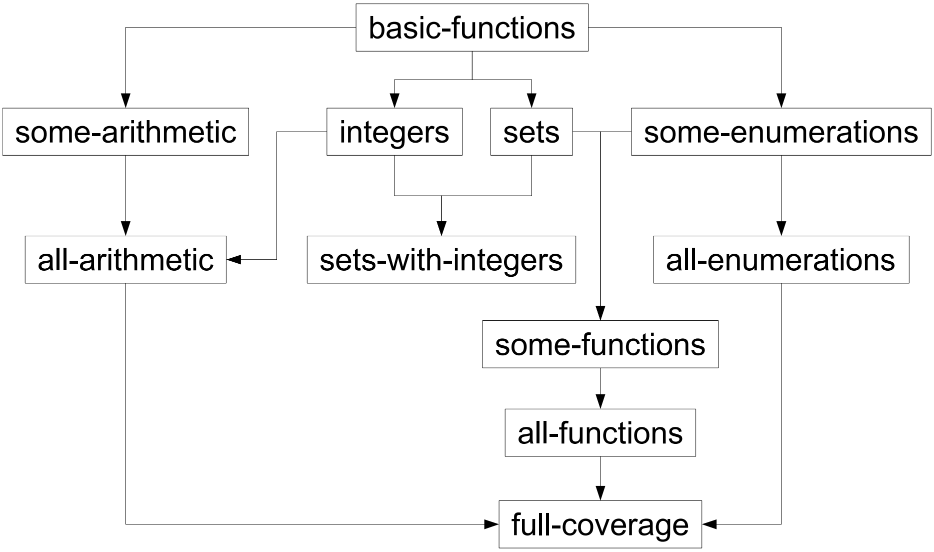

Figure 2. Testing strategies are partially ordered

we will not mention the logical coverage unless necessary.

- SOME-ENUMERATIONS covers all enumerated types. In this way it guarantees that sensible values declared in enumerated types will all be exercised at least once.
- ALL-ENUMERATIONS is a stronger form of SOME-ENUMERATIONS since it not only considers enumerated types but also all extensional sets defined in the specification.
- INTEGERS, instead of tackling enumerations, looks for integer overflows by covering all the integer expressions of the specification. It requires at least five test cases for each integer expression by applying NR.
- SOME-ARITHMETIC covers the arithmetic operators with the defined standard partitions.
- ALL-ARITHMETIC puts together INTEGERS and SOME-ARITHMETIC.
- SETS covers all the set operators with the defined standard partitions.
- SETS-WITH-INTEGERS puts together INTEGERS and SETS.
- SOME-FUNCTIONS combines SETS and SOME-ENUMERATIONS, thus covering some interesting values (i.e. constants of enumerations) and all the set operators.

Table 1. Domain partition rules used in testing strategies

Strategy	Rules
BASIC-FUNCTIONS	DNF
SOME-ENUMERATIONS	DNF, FT
ALL-ENUMERATIONS	DNF, FT, ISE
INTEGERS	DNF, NR
SOME-ARITHMETIC	DNF, SP over arithmetic operators
ALL-ARITHMETIC	DNF, NR, SP over arithmetic operators
SETS	DNF, SP over set and relational operators
SETS-WITH-INTEGERS	DNF, NR, SP over set and relational operators
SOME-FUNCTIONS	DNF, FT, SP over set and relational operators
ALL-FUNCTIONS	DNF, FT, SP
FULL-COVERAGE	DNF, FT, SP, NR, ISE

- ALL-FUNCTIONS combines SETS, SOME-ARITHMETIC and SOME-ENUMERATIONS covering in this way the essential elements appearing in the specification.
- FULL COVERAGE combines ALL-FUNCTIONS, INTEGERS and ALL-ENUMERATIONS.

Table 1 lists the set of partitioning rules that are applied for each strategy. Note that, due to the way recursive domain partition is performed (i.e. by logical conjunction, see Section 2), it is not relevant the order in which rules are applied. This also explains the partial order used to build the graph of Figure 2. In effect, if S_1 and S_2 are two testing strategies such that there is an arrow pointing from S_1 to S_2, then S_2 will produce at least the same set of test conditions than S_1. This is so because all the partitioning rules that are applied by S_1 are also applied by S_2 plus some more. In general, these extra partitioning rules not necessarily produce new test conditions although they will produce them in many cases. For example, ALL-FUNCTIONS will not produce different results than SETS if there are no variables of enumerated types; but if there are, then it will produce more test conditions.

When we partitioned the input space declared in (2) we applied the standard partition of $\lhd$ only to IS_1, and not to IS_2. The reason to proceed in this way is that it is unlikely that an error in the implementation of $\lhd$ can be revealed when the symbol to be removed is not in the symbol table (i.e. IS_2). In effect, a possible pseudo-code implementation of (1) might be:

```
if ($s?$ is an element of the symbol table)
then remove $s?$ from the symbol table
```

where "$s?$ is an element of the symbol table" is the implementation of $s? \in \operatorname{dom} st$ and "remove $s?$ from the symbol table" is the implementation of $s? \lhd st$. The inner block of the if – then sentence may entail several lines of code depending on the data structure defined to hold the symbol table. Note that if $s? \notin \operatorname{dom} st$ nothing is done—i.e., there is no else branch. Therefore, the implementation of $s? \lhd st$ will be tested only if $s? \in \operatorname{dom} st$, so it makes no sense to exercise the implementation in different ways when $s? \notin \operatorname{dom} st$. This is equivalent to not further partition IS_2.

In summary, testing strategies will produce good coverage with a minimum number of test cases if recursive domain partition is applied only to some test specifications. Then, every testing strategy defined above applies domain partition as follows:

- Consider that SP is applied to the expression $\alpha \Box \beta$, where $\Box$ is any mathematical operator and α and β are two subexpressions. If $\alpha \Box \beta$ is part of the precondition of the operation, then SP is applied to all test conditions where some variable in α or β is present. If $\alpha \Box \beta$ is part of the postcondition, then SP is applied to all test conditions whose predicates imply the precondition that leads to that postcondition.
- FT and NR are applied to all test conditions where the variable or expression being considered is present.
- ISE is applied to all test conditions where any of $expr, expr_1, \dots, expr_n$ is present.

Consider for a moment the testing strategy named SETS. SETS applies SP to all the set operators appearing in the operation being considered. This definition, however, is a little bit ambiguous because a set operator can be used in different expressions in the same model operation. Will SETS apply SP to all the expressions or just to one of them (for each set operator)? Would it worth to apply SP to the same operator several times because it appears in several different expressions? The answer is it depends, as the following example illustrates.

Example 2 *Consider that in a particular implementation sets are implemented as singly-linked lists and there is a subroutine, called* union*, that takes two sets (i.e. singly-linked lists) and returns their union. Now say that* union *is called several times with different real parameters from another subroutine, named* caller*. At specification level, these calls would be represented as different set-union expressions. Should* SETS *apply SP over all these set-union expressions? As concerns to the correctness of* union *it is a waste of resources to exercise this subroutine with the same set of test cases generated from the set-union expressions appearing in* caller*. In fact, testing* union *should be performed from its own specification and not from the places where it is called.*

However, as concerns to the correctness of caller*, it should be tested whether these set-union expressions are correctly implemented. In turn, this means to test whether the implementation-level operands of these expressions correspond to the specification-level ones.*

Table 2. Expressions considered for testing intensity

Partitioning rule	Expressions
FT	variables of enumerated types
SP	mathematical operators and the expressions where they appear
NR	arithmetic expressions
ISE	expressions of the form $expr \in \{expr_1, \ldots, expr_n\}$

For instance, if at specification level we have $C = A \cup B$ it must be implemented as $\mathsf{union}([\![A]\!], [\![B]\!], [\![C]\!])$ *where $[\![\alpha]\!]$ means the implementation of α. It would be wrong to implement it as, for example, any of the following:* $\mathsf{union}([\![D]\!], [\![B]\!], [\![C]\!])$, $\mathsf{union}([\![A]\!], [\![B]\!], [\![D]\!])$, $\mathsf{otherFunc}([\![A]\!], [\![B]\!], [\![C]\!])$, *etc. So, in essence, all that should be tested is whether the set-union expressions are implemented as calls to* union *with the right parameters. Applying SP for these expressions is too much because it generates tests cases that test more than that.*

Nevertheless, if each set-union expression in caller *is implemented with its own code (i.e. not calling a general subroutine like* union*), then a good set of test cases for each such expression should be generated. Then, in this case, it would be advisable for* SETS *to apply SP over all the set-union expressions in the specification of* caller.

The same analysis done for SETS can be applied to all testing strategies. Hence, our proposal for testing strategies includes a parameter that we call *coverage level*. Coverage level is a positive natural number whose default value is 1. In general, the grater this value, the greater the number of test conditions that will be generated, and the greater the coverage produced. The number of test conditions that will be generated depends on the number of expressions of a particular kind that are present in the operation whose input domain is being partitioned. Table 2 shows the kind of expressions considered for each partitioning rule. Then, the greater the coverage level, the greater the number of such expressions that are considered during the partitioning process. If a strategy is defined by more than one partitioning rule the product of the number of the corresponding expressions defines the coverage level for the strategy.

Example 3 *If A is an operation that has n different expressions with a mathematical operator and m variables of enumerated types, then the maximum coverage level for* ALL-FUNCTIONS *when applied to A is $n * m$. Hence, if the user applies* ALL-FUNCTIONS *to A with a testing intensity k (with $1 < k \leq n * m$), SP and FT will be applied k times each (to different expressions and different enumerated variables) if that is possible (otherwise one of the partitioning rules will be applied more times until summing up k). On the other hand, if* ALL-FUNCTIONS *is invoked with a coverage level of 1 (default value), then SP and FT will be applied one time* *each to two expressions chosen from the available ones.*

If the partitioning rules are not applied on all the possible expressions, then the specific expressions over which they are applied should be chosen based on the empirical knowledge about the effectiveness of partitioning rules to find errors. This knowledge suggests that operators such as $\oplus$ and $\lhd$ should be favored over $\cup$ and $\cap$, that expressions involving longer sub-expressions should be preferred over those with simpler ones, and that larger enumerated types or set extensions should win over smaller ones.

5 Testing Strategies in Practice

In this section we show a prototype implementation of testing strategies in FASTEST. FASTEST [8] is a MBT tool providing support for the Test Template Framework (TTF) [29]. The TTF is a MBT method that uses Z specifications as models from which test cases are generated. In the TTF and FASTEST partitioning rules are called *testing tactics*. FASTEST is freely available from `http://www.fceia.unr.edu.ar/~mcristia/fastest-1.6.tar.gz`.

As a running example, consider the following Z operation over the symbol table:

$$Update == [st, st' : SYM \pfun \mathbb{Z};\ s? : SYM;\ v? : VAL \mid st' = st \oplus \{s? \mapsto v?\}]$$

Clearly, the specification uses a complex relational operator ($\oplus$) but it also deals with integer numbers. In Z the set of integer numbers is infinite. So when an integer Z variable is implemented, some programming language type is usually chosen (such as `int`, `short`, etc.). Therefore, it would be important to check whether $\oplus$ and the refinement of $\mathbb{Z}$ to, say, `short`, are correctly implemented.

In terms of testing tactics, checking the correct implementation of $\oplus$ and $\mathbb{Z}$ requires to apply SP over $st \oplus \{s? \mapsto v?\}$ and NR on $v?$ with the range $[-32768, 32767]$. Before the implementation of testing strategies, FASTEST's users needed to:

1. Manually analyze the specification to see what testing tactics could be applied;
2. Decide to what test specifications the selected tactics must be applied;
3. Decide over what expressions the selected testing tactics must be applied; and
4. Indicate to FASTEST the corresponding commands with all their parameters

Example 4 *The application of SP and NR as indicated above would require the following commands:*

```
addtactic Update SP \oplus st \oplus \{s? \mapsto v?\}
addtactic Update NR v? \langle -32768, 32767 \rangle
```

With the introduction of testing strategies FASTEST' users only need to have a broad knowledge of the specification. For example, does the specification use set operators? Does it use integer expressions? Does it use enumerated types? And, at the same time, users need to know how much testing can be performed within the available time and budget. All this information will indicate the best testing strategy based on the description of what it tests. After choosing the strategy they just need to issue a command such as the following:

```
applystrategy set-with-integers Update
```

In this case the best tactic is SET-WITH-INTEGERS because it deals with the implementation of set operators and variables whose type is $\mathbb{Z}$. If the implementation of *Update* is available, then FASTEST can even find the right range for NR. In effect, FASTEST also implements test case refinement [10], which requires from engineers to indicate what implementation variables refine each specification variable. Therefore, through this information, FASTEST can find the implementation variable corresponding to $v?$, what is the type of this variable in the implementation, what is the programming language, and, finally, what are the limits for that type in that language.

The `applystrategy` command shown above would generate 20 tests specifications, two of which are the following:

$$
\begin{array}{l}
Update^{NR}_{18} == \\
\quad [Update^{VIS} \mid \\
\qquad st \neq \emptyset \land \operatorname{dom} st \cap \operatorname{dom}\{s? \mapsto v?\} = \emptyset \land v? > -32768 \land v? < 32767] \\
Update^{NR}_{20} == \\
\quad [Update^{VIS} \mid \\
\qquad st \neq \emptyset \land \operatorname{dom} st \cap \operatorname{dom}\{s? \mapsto v?\} = \emptyset \land v? > 32767]
\end{array}
$$

After issuing an `applystrategy` command FASTEST's users need to call the test case generation command (i.e. `genalltca`). In other words, `applystrategy` only generates test specifications.

Coverage level has been implemented as a parameter for the `applystrategy` command; if it is not set the default value (1) is taken.

FASTEST also features a scripting language that allows engineers to define new testing strategies based on the available testing tactics. For example, the following script implements a simple testing strategy that applies SP over all the operators of a given Z operation:

```
STRATEGY FullSP
  FOREACH( e : getSPExpressions(OP) )
    addtactic OP SP e.op e
  END
  genalltt
  prunett
END
```

where `FullSP` is the name of the strategy, `OP` is a built-in variable holding the name of the operation passed to the `applystrategy` command, `genalltt` is a FASTEST command that applies the testing tactics queued with `addtactic`, and `prunett` is another FASTEST command that eliminates unsatisfiable test specifications [8]. `FOREACH` is simple looping instruction of the scripting language which iterates over all the values given by the expression between parenthesis. In turn, these expressions can access internal data structures in a rather abstract fashion. In this example, the `getSPExpressions()` function is used. This function returns all the expressions where a mathematical operator is used. The returned expressions are objects that can be easily accessed—for example, `e.op` returns the operator used in `e`.

All of the testing strategies defined so far in FASTEST have been implemented using this scripting language.

6 Discussion

There are several MBT methods that use specification languages whose models are complex formulas over some logic and mathematical theories [3, 16, 18, 19, 21, 6, 28]. Some of these methods also rely on domain partition. None of them show how coverage criteria can be defined directly over set-based specifications. These methods may be benefited by the ideas proposed in this paper. Ammann et al. define coverage criteria for logic and state-based specifications but they do not take into account complex mathematical theories such as set theory [4, 2]. As we have said in the introduction, some MBT methods [18, 19] think of test cases as sequences of operations that execute the implementation. In these cases, users can extract these sequences by traversing an automaton, which is derived from the specification. In turn, they can apply different testing criteria that traverse the automaton in different ways, thus generating different sets of sequences. This concept is somewhat similar to testing strategies as proposed in this paper, although perhaps strategies provide more intuitive coverage criteria for mathematical specifications, such as those written in Z or B.

The concept of testing strategy came form the realization that domain partition rules provide only local or partial coverage over the specification. Strategies are applicable to the whole specification like path- or data-based coverage criteria from structural testing.

Besides, testing strategies embody the experience and knowledge gained after applying MBT to several projects and case studies [12, 7]. In this sense, the concept of testing strategy is not the mere assembly of partitioning rules nor their blind application to each statement of the specification. Strategies really relieve testers from some non-trivial analysis by, as we have said, implementing known testing heuristics. For example, SOME-ARITHMETIC applies SP only over arithmetic operators because its focus is on the correct implementation of arithmetics. Similarly, a structural criterion like condition-coverage [14] tests conditions not in its most general way. As another example, BASIC-FUNCTIONS, SOME-ENUMERATIONS and ALL-ENUMERATIONS provide a minimal coverage like statement or branch coverage.

Observe that testing strategies do not change the underlying theory nor the basic techniques of MBT. This implies that engineers can combine strategies and domain partition to produce test sets as they wish. The partial order that organizes strategies can be extended or modified as new partitioning rules are created or more insight about the existing ones is gained.

The presentation made in Section 2 is essentially that of the TTF. However, in this paper we go one step further by defining some criteria that generate test cases by covering the set-based specification in different ways. Also note that the presentation made in this paper is more general than that made by Stocks and Carrington since we do not rely on any particular notation.

Although Figure 2 should help users to select the right strategy for each operation, testing strategies should be accompanied by descriptive cards listing their main features in terms of the coverage they provide, the kind of test cases they generate or the aspects they exercise. The vocabulary of these cards should be that of test engineers rather than that of the formal methods or academic testing communities. More technical (formal) information can be provided for advanced users.

Coverage level is the most abstract and general way we could think of to allow for different levels of coverage. Indeed, if engineers want that some testing strategy applies all its partitioning rules to all the possible expressions, then they can set the coverage level to its maximum. This would be the case for the second implementation described in Example 2. On the contrary, if they want less test cases they can set coverage level to one, which will force the testing strategy to apply its partitioning rules just once. Another possible approach could be to define variants of the testing strategies concerning more fine grained coverage levels. For example, SETS can be defined as to apply SP *only once* to each set operator and a new strategy, say ALL-SETS, can be defined as applying SP to *to all the different expressions* where a set operator is present.

7 Conclusions

The main conclusion we can draw from this paper is that it is possible to define coverage criteria for MBT methods based on logic and mathematics that play a similar role to path- or data-based coverage criteria in structural testing. This gives a very abstract and testing-oriented view of MBT methods based on this kind of specifications. Furthermore, the partial order that can be defined among testing strategies allows testers to choose the right strategy by what will be tested at the implementation level rather than by how an input domain is partitioned. The partial order of testing strategies can be modified and extended with new strategies as they are defined or improved.

Therefore, testing strategies can help test engineers in two regards: a) they simplify, automate and hide domain partition; and b) they provide a more intuitive and abstract way for test case generation from set-based specifications.

In the future we plan to finish the implementation of all the testing strategies described here; to write descriptive cards that help testers to informally understand what each strategy will test and the relation between them; and to explore whether some partitioning rules that have not be considered so far [9], can be used to define new testing strategies.

8 *

References

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] Paul Ammann, A. Jefferson Offutt, and Hong Huang. Coverage criteria for logical expressions. In *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA*, pages 99–107. IEEE Computer Society, 2003.

[3] Paul Ammann and Jeff Offutt. Using formal methods to derive test frames in category-partition testing. In *Compass'94: 9th Annual Conference on Computer Assurance*, pages 69–80, Gaithersburg, MD, 1994. National Institute of Standards and Technology.

[4] Paul Ammann, Jeff Offutt, and Wuzhi Xu. Coverage criteria for state based specifications. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 118–156. Springer, 2008.

[5] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of Test Sequences from Formal Specifications: GSM 11-11 Standard Case Study. *International Journal of Software Practice and Experience*, 34(10):915–948, 2004.

[6] Simon Burton. Automated Testing from Z Specifications. Technical report, Department of Computer Science – University of York, 2000.

[7] Maximiliano Cristiá, Pablo Albertengo, Claudia S. Frydman, Brian Plüss, and Pablo Rodríguez Monetti. Applying the Test Template Framework to aerospace software. In James L. Rash and Christopher Rouff, editors, *SEW*, pages 128–137. IEEE Computer Society, 2011.

[8] Maximiliano Cristiá, Pablo Albertengo, Claudia S. Frydman, Brian Plüss, and Pablo Rodríguez Monetti. Tool support for the Test Template Framework. *Softw. Test., Verif. Reliab.*, 24(1):3–37, 2014.

[9] Maximiliano Cristiá and Claudia S. Frydman. Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions. In John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *ABZ*, volume 7316 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2012.

[10] Maximiliano Cristiá, Diego Hollmann, Pablo Albertengo, Claudia S. Frydman, and Pablo Rodríguez Monetti. A language for test case refinement in the Test Template Framework. In Shengchao Qin and Zongyan Qiu, editors, *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, pages 601–616. Springer, 2011.

[11] Maximiliano Cristiá, Gianfranco Rossi, and Claudia S. Frydman. {log} as a test case generator for the Test Template Framework. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2013.

[12] Maximiliano Cristiá, Valdivino Santiago, and N.L. Vijaykumar. On comparing and complementing two MBT approaches. In *Test Workshop (LATW), 2010 11th Latin American*, pages 1–6, 2010.

[13] Jeremy Dick and Alain Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[14] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering (2nd ed.).* Prentice Hall, 2003.

[15] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Víctor A. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Softw. Test., Verif. Reliab.*, 21(1):55–71, 2011.

[16] P. A. V. Hall. Towards testing with respect to formal specification. In *Proc. Second IEE/BCS Conference on Software Engineering*, number 290 in Conference Publication, pages 159–163. IEE/BCS, July 1988.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[18] R. M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using Statecharts and Z. *Information and Software Technology*, 43(2):137–149, February 2001.

[19] Robert M. Hierons. Testing from a Z specification. *Software Testing, Verification & Reliability*, 7:19–33, 1997.

[20] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

[21] Hans Martin Hörcher and Jan Peleska. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4:309–327, 1995.

[22] Daniel Jackson. Alloy: A logical modelling language. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, page 1. Springer, 2003.

[23] Cliff B. Jones. *Systematic Software Development Using VDM (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[24] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. *CoRR*, abs/1303.1006, 2013.

[25] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[26] Manoranjan Satpathy, Michael Leuschel, and Michael Butler. ProTest: An automatic test environment for B specifications. *Electronic Notes in Theroretical Computer Science*, 111:113–136, January 2005.

[27] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[28] Susan Stepney. Testing as abstraction. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM*, volume 967 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 1995.

[29] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[30] Mohammad Saeed Abou Trab, Michael Brockway, Steve Counsell, and Robert M. Hierons. Testing real-time embedded systems using timed automata based approaches. *Journal of Systems and Software*, 86(5):1209–1223, 2013.

[31] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[32] J. Zander, I. Schieferdecker, and P.J. Mosterman. *Model-based Testing for Embedded Systems*. Computational Analysis, Synthesis, and Design of Dynamic Systems Series. CRC Press, 2012.