Coverage Criteria for Logical Specifications

Maximiliano Cristiá[1] **, Joaquín Cuenca**[1] **, Claudia Frydman**[2]

[1]CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar, joacuenca@gmail.com

[2]LSIS-CIFASIS, Marseille, France
claudia.frydman@lsis.org

Abstract. *Model-based testing (MBT) studies how to generate test cases 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 the Z notation) it may be more natural or intuitive to apply coverage criteria directly over the formula. In this paper we propose a set of coverage criteria for logical specifications based on domain partition. We call them* testing strategies. *Testing strategies play a similar role to path- or data-based coverage criteria. 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.*

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 lot of 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 aims at automatically generating test cases from the analysis of a model of the system under test (SUT) [22]. MBT methods have achieved impressive theoretical and practical results in recent years such as [13, 8, 21, 15, 23], 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 [17] 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. [11]. In this case a Z specification [18] is accompanied by a Statechart [10] 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.

As can be seen, even when the model of the SUT is (essentially) a logical formula (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 logical formula. These criteria are defined by conveniently assembling together rules that indicate how to partition an input domain. Our criteria do not need an automaton because they analyze the structure, semantics and types of a logical formula. These criteria can be organized in a partial order to help users to select the most appropriate for each project.

We also show how these criteria have been implemented in FASTEST [5], a tool that supports the Test Template Framework (TTF) [20] which, in turn, is a MBT method for the Z notation.

The paper starts by introducing the concept of domain partition in Section 2. Testing strategies are introduced in Section 3 where they are accommodated in partial order and a prototype implementation is shown. We also discuss the contribution of this paper along with its conclusions in Section 4.

2. Domain Partition

In this section we show some rules for domain partition that are already available. These rules are later combined to define criteria that provide different levels of coverage of the logical specification to which they are applied.

Consider the following logical formula as an example of some specification:

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

where $st : SYM \pfun \mathbb{Z}$ is a partial function; SYM is a given, underspecified set; $s? : SYM$ 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. This formula formalizes the elimination of a symbol ($s?$) from a symbol table (st) that associates elements of SYM with integer numbers. In other words, (1) represents 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 build an automaton from (1), as suggested by some MBT methods [17, 12, 11], because we have only one transition. On the other hand, the implementation of (1) is not trivial because, for instance, it seldom will 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 tested? An answer given by some methods is to use domain partition [20, 2]. First, the input domain or space of the specification is defined, then it is partitioned in subsets called *test conditions* and finally one element of each test condition is taken as a test case. For example, the input space of (1) is defined as:

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

$$R = \emptyset$$
$$R \neq \emptyset \wedge S = \emptyset$$
$$R \neq \emptyset \wedge S = \operatorname{dom} R$$
$$R \neq \emptyset \wedge S \neq \emptyset \wedge S \subset \operatorname{dom} R$$
$$R \neq \emptyset \wedge S \neq \emptyset \wedge S \cap \operatorname{dom} R = \emptyset$$
$$R \neq \emptyset \wedge S \neq \emptyset \wedge S \cap \operatorname{dom} R \neq \emptyset \wedge \operatorname{dom} R \subset S$$
$$R \neq \emptyset \wedge S \neq \emptyset \wedge S \cap \operatorname{dom} R \neq \emptyset \wedge \neg\ (\operatorname{dom} R \subseteq S) \wedge \neg\ (S \subseteq \operatorname{dom} R)$$

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

and we can partition it by taking the precondition of each disjunct in (1):

$$IS_1 = \{IS \mid s? \in \operatorname{dom} st\} \tag{3}$$
$$IS_2 = \{IS \mid s? \notin \operatorname{dom} st\} \tag{4}$$

With this partition we can define two test cases:

$$TC_1 = \{IS_1 \mid st = \{(s_1, 3)\} \wedge s? = s_1\} \tag{5}$$
$$TC_2 = \{IS_2 \mid st = \emptyset \wedge s? = s_1\} \tag{6}$$

However, are these test cases enough? 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 these test cases will not test whether the iteration over the list is well 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.

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. Then, we can substitute *R* by *st* and *S* by $\{s?\}$ and use this to partition IS_1 as follows:

$$IS_1^1 = \{IS_1 \mid st = \emptyset\} \tag{7}$$
$$IS_1^2 = \{IS_1 \mid st \neq \emptyset \wedge \{s?\} = \emptyset\} \tag{8}$$
$$IS_1^3 = \{IS_1 \mid st \neq \emptyset \wedge \{s?\} = \operatorname{dom} st\} \tag{9}$$
$$IS_1^4 = \{IS_1 \mid st \neq \emptyset \wedge \{s?\} \neq \emptyset \wedge \{s?\} \subset \operatorname{dom} st\} \tag{10}$$
$$IS_1^5 = \{IS_1 \mid st \neq \emptyset \wedge \{s?\} \neq \emptyset \wedge \{s?\} \cap \operatorname{dom} st = \emptyset\} \tag{11}$$
$$IS_1^6 = \{IS_1 \mid st \neq \emptyset \wedge \{s?\} \cap \operatorname{dom} st \neq \emptyset \wedge \operatorname{dom} st \subset \{s?\}\} \tag{12}$$
$$IS_1^7 = \{IS_1 \mid st \neq \emptyset \wedge \{s?\} \cap \operatorname{dom} st \neq \emptyset \wedge \neg\ \operatorname{dom} st \subseteq \{s?\} \wedge \neg\ \{s?\} \subseteq \operatorname{dom} st\} \tag{13}$$

Note 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. Also observe that $s \in \operatorname{dom} st$ is implicitly conjoined to every predicate of (7)-(13).

Surely more test cases are needed to test the implementation of (1) but we think we have make it clear that logical specifications contain enough information as to derive a good test case set without necessarily resorting to an automaton.

2.1. More Partitioning Rules

Free Types (FT) helps to partition an input domain when variables whose type is enumerated are used. If x is a variable of an enumerated type, T, FT generates test conditions whose characteristic predicates are of the form $x = val$ for each $val \in T$. In this way, the FT 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. For example, 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 Z specification as an expression of type $\mathbb{Z}$. For instance, a potential C implementation of the symbol table described in the previous section might be as follows: *st* can be a simply-linked list of nodes defined as:

```
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 expression $st(s?)$. In this case NR would generate, for example, $st(s?) < -32768$ 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.

3. 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 unrelated partitioning techniques. 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 (logic-)specification-based covering criteria.

Having the seminal work of Rapps and Weyuker [16] as an inspiration, we organized the 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:

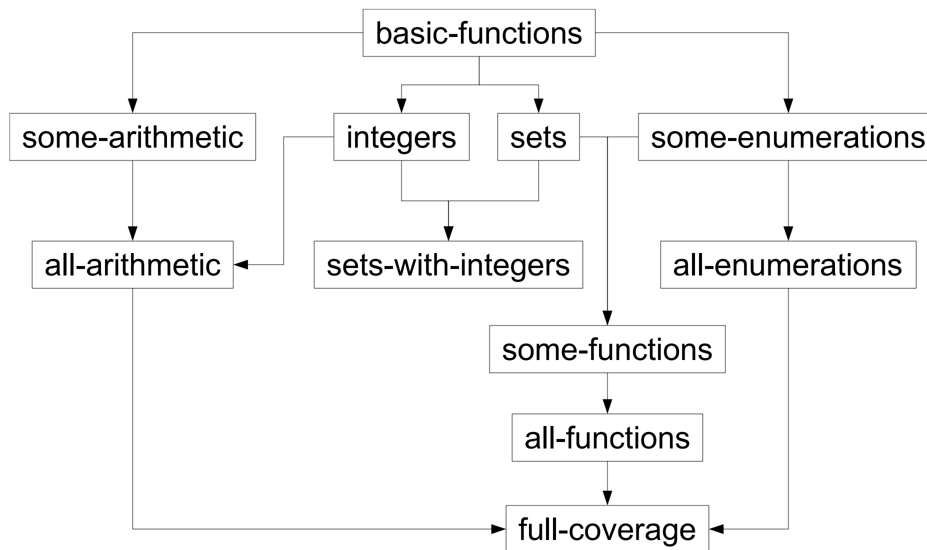

Figure 2. Testing strategies are partially ordered

- 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 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 it 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.
- 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 domain partition rules that are applied for each strategy. Note that, due to the way 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 domain partition rules that are applied by S_1 are also applied by S_2 plus some more. In general, these extra

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

Table 1. Domain partition rules used in testing strategies

domain partition 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 st*

where "$s?$ *is an element of the symbol table*" is the implementation of $s? \in \operatorname{dom} st$ and "*remove* $s?$ *from st*" is the implementation of $s? \lhd st$, which 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. 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 partition IS_2.

In summary, testing strategies will produce good coverage with a minimum number of test cases if 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 Z 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 $expr, expr_1, \ldots, expr_n$ is present.

3.1. Testing Strategies in Practice

In this section we show how we have implemented in FASTEST the concept of testing strategy. FASTEST [5] is a MBT tool providing support for the Test Template Framework (TTF) [20]. The TTF is a MBT method that uses Z specifications as models from which test cases are generated. The tool if freely available from `http://www.fceia.unr.edu.ar/~mcristia/fastest-1.6.tar.gz`.

FASTEST already implements the concept of domain partition by so-called *testing tactics*. Testing tactics are applied to Z schemas. FASTEST implements testing strategies as structural testing tools implement coverage criteria. Therefore, FASTEST's users need to indicate what testing strategy they want to use and the tool applies testing tactics (i.e. domain partition) according to the strategy definition. Different strategies can be chosen for different operations in the specification. Strategies completely hide from users the complexities of domain partition.

As an 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 restriction to say, `short`, are correctly implemented. A testing strategy that covers all these situations is SET-WITH-INTEGERS. If this strategy is used, FASTEST generates 20 tests specifications. Due to space restrictions only two of them are shown below:

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

FASTEST also features a scripting language that allows engineers to define new testing strategies that later are automatically applied.

4. Discussion and Conclusions

There are several MBT methods that use specification languages whose models are complex formulas over some logic and mathematical theories [2, 9, 11, 12, 14, 3, 19]. Some of these methods also rely on domain partition. None of them show how coverage criteria can be defined directly over the logical specifications. These methods may be benefited by the ideas proposed in this paper. As we have said in the introduction, some MBT methods [11, 12] 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 logical specifications.

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 [6, 4]. 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 [7] 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 logical 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.

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. This partial order can be modified and extended with new strategies as they are defined or improved.

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, can be used to define new strategies.

References

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

[2] Ammann, P., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: Compass'94: 9th Annual Conference on Computer Assurance. pp. 69–80. National Institute of Standards and Technology, Gaithersburg, MD (1994), `citeseer.ist.psu.edu/ammann94using.html`

[3] Burton, S.: Automated Testing from Z Specifications. Tech. rep., Department of Computer Science – University of York (2000)

[4] Cristiá, M., Albertengo, P., Frydman, C.S., Plüss, B., Monetti, P.R.: Applying the Test Template Framework to aerospace software. In: Rash, J.L., Rouff, C. (eds.) SEW. pp. 128–137. IEEE Computer Society (2011)

[5] Cristiá, M., Albertengo, P., Frydman, C.S., Plüss, B., Rodríguez Monetti, P.: Tool support for the Test Template Framework. Softw. Test., Verif. Reliab. 24(1), 3–37 (2014)

[6] Cristiá, M., Santiago, V., Vijaykumar, N.: On comparing and complementing two MBT approaches. In: Test Workshop (LATW), 2010 11th Latin American. pp. 1–6 (2010)

[7] Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of software engineering (2nd ed.). Prentice Hall (2003)

[8] Grieskamp, W., Kicillof, N., Stobie, K., Braberman, V.A.: Model-based quality assurance of protocol documentation: tools and methodology. Softw. Test., Verif. Reliab. 21(1), 55–71 (2011)

[9] Hall, P.A.V.: Towards testing with respect to formal specification. In: Proc. Second IEE/BCS Conference on Software Engineering. pp. 159–163. No. 290 in Conference Publication, IEE/BCS (Jul 1988)

[10] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)

[11] Hierons, R.M., Sadeghipour, S., Singh, H.: Testing a system specified using Statecharts and Z. Information and Software Technology 43(2), 137–149 (February 2001), `http://dx.doi.org/10.1016/S0950-5849(00)00145-2`

[12] Hierons, R.M.: Testing from a Z specification. Software Testing, Verification & Reliability 7, 19–33 (1997)

[13] Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. 41(2), 1–76 (2009)

[14] Hörcher, H.M., Peleska, J.: Using Formal Specifications to Support Software Testing. Software Quality Journal 4, 309–327 (1995)

[15] Peleska, J.: Industrial-strength model-based testing - state of the art and current challenges. CoRR abs/1303.1006 (2013)

[16] Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE '82: Proceedings of the 6th international conference on Software engineering. pp. 272–278. IEEE Computer Society Press, Los Alamitos, CA, USA (1982)

[17] Satpathy, M., Leuschel, M., Butler, M.: ProTest: An automatic test environment for B specifications. Electronic Notes in Theroretical Computer Science 111, 113–136 (January 2005)

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

[19] Stepney, S.: Testing as abstraction. In: Bowen, J.P., Hinchey, M.G. (eds.) ZUM. Lecture Notes in Computer Science, vol. 967, pp. 137–151. Springer (1995)

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

[21] Trab, M.S.A., Brockway, M., Counsell, S., Hierons, R.M.: Testing real-time embedded systems using timed automata based approaches. Journal of Systems and Software 86(5), 1209–1223 (2013)

[22] Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)

[23] Zander, J., Schieferdecker, I., Mosterman, P.: Model-based Testing for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems Series, CRC Press (2012), http://books.google.com.ar/books?id=fzgzNW_alD0C