Extending the Test Template Framework to Deal With Axiomatic Descriptions, Quantifiers and Set Comprehensions

Maximiliano Cristiá[1] and Claudia Frydman[2]

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

Abstract. The Test Template Framework (TTF) is a method for model-based testing (MBT) from Z specifications. Although the TTF covers many features of the Z notation, it does not explain how to deal with axiomatic descriptions, quantifiers and set comprehensions. In this paper we extend the TTF so it can process specifications including these features. The techniques presented here may be useful for other MBT methods for the Z notation or for other notations such as Alloy and B, since they use similar mathematical theories.

1 Introduction

The Test Template Framework (TTF) is a model-based testing (MBT) method [1, 2], used mainly for unit testing. MBT is a well-known technique aimed at testing software systems by analysing a formal model [3, 4]. MBT approaches start with a formal model or specification of the software, from which test cases are generated. These techniques have been developed and applied to models written in different formal notations such as Z [1], finite state machines and their extensions [5], B [6], algebraic specifications [7], and so on. The fundamental hypothesis behind MBT is that, as a program is correct if it satisfies its specification, then the specification is an excellent source of test cases.

Our group was the first in providing tool support for the TTF by implementing Fastest [8–10], and in extending the TTF beyond test case generation [11, 12]. Furthermore, we have applied Fastest and the TTF to several industrial-strength case studies [8, 13, 14]. The tool greatly automates tactic application, testing tree generation, testing tree simplification, and test case generation.

In 2008 we wrote a Z specification [14] of a significant portion of the ECSS-E-70-41A aerospace standard [15]. This is a medium-sized specification comprising 74 pages and more than 2,000 lines of Z. It is the largest Z specification we have written so far to test and validate Fastest. As a matter of comparison, the Tokeneer specification has only 46 lines more, while it is recognized as a full-fledged, industrial-strength formal specification [16]. The ECSS-E-70-41A formal specification comprises the minimum capability sets of 6 of the 16 services described in the standard. The model includes 25 state variables with 16 of

a relational type, of which 6 are higher-order functions and 3 are defined by referencing schema types. It also contains 28 axiomatic descriptions, some of which define operators whose domain are higher-order functions and schema types. To complicate things even more, this specification defines a number of set comprehensions and lambda expressions that influence critical outputs—for example, the report of housekeeping data of a satellite sent to ground upon request. Finally, some operations include quantified formulas.

Axiomatic descriptions, quantified formulas and set comprehensions were not considered in the original presentation of the TTF nor in Fastest. In this paper, we propose some techniques within the philosophy of the TTF and preserving a good deal of automation that extend the TTF so it can process specifications including these features. Currently, Fastest provides limited tool support for some classes of axiomatic descriptions—those referred as classes $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{O}$ in Sect. 3—and it implements testing tactics for quantified formulas—those referred as WEQ, SEQ and UQ in Sect. 4. Therefore, so far, we have only been able to manually apply the techniques presented in this paper to the ECSS-E-70-41A formal specification—and automatically to some toy examples. Given that these techniques are aligned with the TTF, their full implementation will preserve the degree of automation currently featured by Fastest.

The paper is structured as follows. Section 2 describes the motivations for extending the TTF. The solution we propose for axiomatic descriptions is based on classifying them according to their intended meaning. Hence, in Sect. 3 we present a taxonomy of axiomatic descriptions and how each category should be processed. Section 4 focuses on the problem posed by quantifications, and Sect. 5 on set comprehensions and lambda expressions. Finally, in Sect. 6 we present our concluding remarks.

2 Some Extensions to the Test Template Framework

The TTF and Fastest have been thoroughly presented in many papers [2, 1, 8, 9]. In this section we focus on some difficulties appearing in the TTF when the Z specification being analysed includes axiomatic descriptions, quantifications or set comprehensions. Here we treat the TTF and Fastest as synonyms.

Given a Z specification, users have to select those operation schemas for which they want to generate test cases. As with other MBT methods, the TTF first generates test cases at the specification level, that are later refined to test the implementation corresponding to that specification [12]. In this paper we work only at the specification level. For each selected schema users indicate a set of testing tactics to be applied to it. The first testing tactic partitions the input space of the operation into a set of test specifications—i.e. test conditions or test objectives [3]. The second testing tactic partitions one or more of these test specifications, into more test specifications. The other testing tactics continue with this process. The net effect is a progressive partition of the input space of the operation into test specifications that are more restrictive than the previous ones. A test case is a witnesses satisfying the predicate of a leaf test specification.

Test specifications are Z schemas like the following one:

$$VerifyCmd_4^{SP} == [VerifyCmd_1^{DNF} \mid cmd? \in checksum \land proc1 \neq \emptyset \land proc2 \neq \emptyset \land proc1 \cap proc2 = \emptyset]$$

where *VerifyCmd* is the name of an operation selected by the user; $VerifyCmd_1^{DNF}$ is the test specification that was partitioned by applying the Standard Partitions (SP) testing tactic; *cmd*?, *proc*1 and *proc*2 are input and state variables declared in *VerifyCmd*; and *checksum* is the following axiomatic description:

$$checksum : \mathbb{P}\, FRAME$$

where *FRAME* is a given type.

Fastest generates all the test specifications automatically once users have indicated what testing tactics they want to apply to operations. Since testing tactic application means, essentially, conjoining predicates, it is not unusual to find unsatisfiable test specifications. These test specifications must be eliminated [9]. For the remaining ones, at least one test case must be generated. A test case for a given test specification is a Z schema restricting all the free variables to take one and only one value. In any MBT method, this process is intended to be as automatic as possible as hundreds of test specifications may be generated for a single specification. Fastest implements a sort of satisfiability algorithm for a significant portion of the Z Mathematical Toolkit (ZMT), that, according to our experiments, in average finds test cases for 80% of the satisfiable test specifications [8].

2.1 Axiomatic Descriptions

At this point some questions arise. Given that the satisfiability of $VerifyCmd_4^{SP}$ depends on the value of *checksum*, when should the algorithm to eliminate unsatisfiable test specifications be run? Is it reasonable for Fastest to automatically bind any value to *checksum*? What if users want an implementation for a particular value for it? Would Fastest generate test cases for that implementation or for any of its family [17, pages 36–38 and 143]? Currently, test cases are generated independently for each test specification—i.e. Fastest asks for an instantiation for each and every test specification. Can it be still done in this way in the presence of axiomatic descriptions like *checksum*? Clearly, two test cases cannot bind different values to *checksum* because they would belong to different members of the family of specifications. What if the specification includes an axiomatic description like the following one?

$$root : USER$$

Is *root* intended to be a constant or a variable? And, what if the specification includes the next one?

$$sum : \operatorname{seq}\mathbb{Z} \to \mathbb{Z}$$

$$sum\langle\rangle = 0$$
$$\forall\, s : \operatorname{seq}\mathbb{Z};\ n : \mathbb{Z} \mid s \neq \langle\rangle \bullet sum(s \frown \langle n \rangle) = n + sum\, s$$

Should a value be generated for *sum*? Or should it be treated entirely different from *checksum* and *root*? Should it be treated as an operation and, thus, test cases have to be generated?

As it can be seen, the inclusion of axiomatic descriptions poses a number of issues to be discussed in order to faithfully extend the TTF.

2.2 Quantified Formulas

Let us turn our attention to quantifications. Z provides a number of operators in the ZMT to avoid explicitly writing quantified formulas. For instance, it is convenient to replace $(\forall x : \operatorname{dom} f \bullet f\, x \neq 0)$ by $0 \notin \operatorname{ran} f$, and $(\forall x : \operatorname{dom} f \mid x \in A \bullet f\, x \neq 0)$ by $0 \notin f(\!|\ A\ |\!)$. Both implicit and explicit quantifications usually lead to loops in the implementation. So it is worth to generate test cases to test these loops by analysing these formulas. In the TTF, mathematical operators are analysed by the Standard Partition (SP) testing tactic. That is, a standard partition can be bound to, say, the $\_(\!|\ \_\ |\!)$ operator such that it will generate test specifications asking for different values of both arguments. For instance, a possible standard partition for $f(\!|\ A\ |\!)$ can be: $f = \emptyset \land A = \emptyset$; $f \neq \emptyset \land A = \emptyset$; $f = \emptyset \land A \neq \emptyset$; $f \neq \emptyset \land A \neq \emptyset \land \operatorname{dom} f \cap A = \emptyset$; and so forth. In other words, each of these partitions will exercise the potential loop implementing the operator in different ways. We would like to follow a similar approach for explicit quantified formulas. That is, we would like to have one or more testing tactics associated to quantifications that would yield test specifications that, in turn, would exercise the corresponding potential loop in different ways. For example, a quantification appearing in the ECSS-E-70-41A formalization is the following[3]:

$$\forall i : \operatorname{dom} sa? \bullet \\ sa?\, i + len?\, i \leq sizes\, m? \land cs?\, i = check(dt?\, i) \land len?\, i \leq \#(dt?\, i)$$

where $sa?$ is of type $\operatorname{seq} \mathbb{N}$, and *check* is an axiomatic description. Therefore, it would be desirable to generate test specifications that would test the implementation with $sa?$'s of different lengths.

Quantified formulas over potentially infinite sets pose a problem for any satisfiability algorithm. Hence, the approach we followed is to generate at least some test specifications where the potentially infinite set is replaced by one or more finite ones. In doing so the quantified formula is equivalent to either an unquantified conjunction or disjunction. But this brings in another issue. The first testing tactic applied by Fastest is Disjunctive Normal Form (DNF) [8]. All the other testing tactics in Fastest conjoin more atomic predicates to a given test specification. Hence, at the end, all test specifications are conjunctions of atomic predicates. Some key algorithms of Fastest rely on all test specifications having that property. Therefore, if we define testing tactics to deal with quantified formulas, they should also write the resulting predicates in DNF.

[3] Some of the names used in the formalization of the ECSS-E-70-41A standard have been changed with respect to the original specification due to space restrictions.

2.3 Set Comprehensions and Lambda Expressions

In the ECSS-E-70-41A formalization we heavily used complex set comprehensions and lambda expressions. For instance, we have the following schema:

$$
\begin{array}{|l}
\hline
PeriodicSampOnce \\
\hline
\Xi Housekeeping;\ \Xi Time \\
sOP : SID \pfun PGMODE \pfun PNAME \pfun PVAL \\
rS : \mathbb{P}\, SID \\
pSO : SID \pfun PNAME \pfun PVAL \\
\hline
rS = \{s : hES \mid (hRD\ s).m = p \land t = hCCI\ s + (hRD\ s).ci * dMI\} \\
pSO = (\lambda\, s : rS \bullet (\lambda\, p : \mathrm{dom}(hRD\ s).ns \mid (hRD\ s).ns\ p = 1 \bullet hSV\ s\ p\ t)) \\
sOP = (\lambda\, s : rS \bullet (\lambda\, m : \{pm\} \bullet pSO\ s)) \\
\hline
\end{array}
$$

which is the simplest one in an operation defined by five others schemas like *PeriodicSampOnce*. Note that rS and pSO are referenced in the definition of sOP. sOP is later assembled with other similar variables declared in the other schemas to produce a single output for the operation. Therefore, the definition of the operation in which this schema participates is, essentially, an extremely complex lambda expression that is bound to an output variable. In summary, the operation has a trivial logical structure, while all its complexity lies inside the lambda expressions and set comprehensions. None of the testing tactics defined in the TTF would produce the desired results since none of them is prepared to work with bound variables. Furthermore, the implementation of these complex expressions will likely be very complex too, thus making it imperative to test it thoroughly.

Therefore, we need one or more testing tactics that generate significant test specifications for this kind of expressions. The approach we followed is to propagate the complexity inside the expressions to the outside, and then apply existing testing tactics. For example, if we have $\{x : X \mid P(x) \lor Q(x) \bullet expr(x)\}$, it can be rewritten as $\{x : X \mid P(x) \bullet expr(x)\} \cup \{x : X \mid Q(x) \bullet expr(x)\}$, making it possible to apply SP to $\cup$.

3 A Taxonomy of Axiomatic Descriptions

In Z, axiomatic descriptions can serve many purposes [17, page 143]. For example, an axiomatic description can be used just to give a name to an integer constant, or it can be used to define a function summing all the components of a sequence of integers. As we have said in Sect. 2.1, in our opinion not all the axiomatic descriptions can be treated in the same way with respect to the TTF. Therefore, we consider that a key step towards their inclusion in the TTF is to define a taxonomy for axiomatic descriptions based on their syntax—aiming at capturing their intended use and semantics. In a second step we define how each category will be processed in the TTF. The ultimate goal is making test case generation as automatic as possible in the presence of axiomatic descriptions.

3.1 Given type constants ($\mathcal{C}$)

If T is a given or basic type and we have:

$$x : T$$

then x is said to be a constant of type T. An example is *root* (Sect. 2.1).

Axiomatic descriptions of this kind are regarded as constants of their corresponding types. Therefore, they will be used as values for variables appearing in test specifications. Two members of $\mathcal{C}$ of the same type will be considered as different constants. For example, if *admin* is an axiomatic description of type *USER*, then $admin \neq root$ holds. However, at the same time, if an operation declares $usr? : USER$, testers can generate test specifications asking for $usr? = root$, $usr? = admin$ and $usr? \notin \{root, admin\}$. For $\mathcal{C}$ no user action is required.

3.2 Synonyms ($\mathcal{S}$)

A synonym is any axiomatic description matching any of the following:

$$\frac{x : T}{x = expr} \qquad \frac{x : T}{\forall\, y : U \bullet x(y) = expr(y)}$$

where T and U are any types and *expr* is any expression. x may depend on some y only if T is a structured type, in which case U is part of T's definition. *expr* may depend on y and, possibly, on other axiomatic descriptions. We call *expr* the definition of x. An example of this kind is the following one taken from the ECSS-E-70-41A formalization:

$$\frac{lastRepVal : (TIME \pfun PVAL) \to PVAL \to \mathbb{N} \to \operatorname{seq} PVAL}{\begin{array}{l}\forall\, h : TIME \pfun PVAL;\ v : PVAL;\ r : \mathbb{N} \bullet \\ \quad lastRepVal\, h\, v\, r = ((\#h - r + 2 \mathinner{..} \#h) \rhd squash\, h) \frown \langle v \rangle\end{array}}$$

Axiomatic descriptions in this category can be treated in two ways:

1. Simply replace the axiomatic descriptions by their definitions when they appear in test specifications. If x is of the quantified form, replace it by its definition substituting its formal parameter by the real one. No user action is needed for $\mathcal{S}$, in this case.
2. Users may want to generate test cases for *expr* as if it were an operation. However, this is not always applicable. For example, it makes sense to do it with *lastRepVal* but it makes no sense with the following one:

   $$\frac{administrators : \mathbb{P}\, USER}{administrators = \{root, admin\}}$$

   In general, this decision must be left to users; Fastest should do as in 1 by default. If the user decides to generate test cases for x, then he/she can use any of the available testing tactics. However, when these axiomatic descriptions appear in a test specification they have to be processed as in 1.

3.3 Equivalences ($\mathcal{E}$)

An equivalence is any axiomatic description matching the following:

$$\begin{array}{|l} x : T \\ \hline \forall\, y : U \bullet P(x, y) \Leftrightarrow Q(y) \end{array}$$

where T and U are any types and P and Q are predicates. Q may depend also on other axiomatic descriptions. We say Q is the definition of x. *falied* is an instance of this category borrowed from the ECSS-E-70-41A formalization:

$$\begin{array}{|l} failed : \mathbb{P}((TIME \pfun PVAL) \times PVAL \times CheckDef) \\ \hline \forall\, h : TIME \pfun PVAL;\ v : PVAL;\ d : CheckDef \bullet \\ \qquad (h, v, d) \in failed \Leftrightarrow avrDelta(lastRepVal\ h\ v\ d.rep) < d.low \end{array}$$

This class is treated as $\mathcal{S}$, only considering that Q is the definition of x.

Note that if we would have defined *failed* as a set comprehension, then it would have fallen in $\mathcal{S}$ thus replacing *failed* for its definition in test specifications. Hence, later, the testing tactics defined in Sect. 5 can be applied. In either way, the expression can be properly treated.

3.4 Inductive definitions ($\mathcal{ID}$)

We say that an axiomatic description is an inductive definition if it has the following form:

$$\begin{array}{|l} x : T \\ \hline \forall\, y_1 : U_1 \bullet x(E_1(y_1)) = expr_1 \\ \ldots\ldots\ldots \\ \forall\, y_n : U_n \bullet x(E_n(y_n)) = expr_n \end{array}$$

where T, $U_1, \ldots, U_n$ are types for which an induction principle is defined—i.e. free types, $\mathbb{N}$, $\mathrm{seq}\, X$ [17, pages 83, 114 and 123], and finite sets [18, page 59]—, $E_1, \ldots, E_n$ are n structurally different expressions of the same inductive type W, and $expr_1, \ldots, expr_n$ are expressions. Any of the quantifiers might be absent in which case the corresponding E expression will be constant. It is assumed that there are no mutually recursive definitions and no definition is infinitely recursive. An element in $\mathcal{ID}$ is *sum* in Sect. 2.1.

As with $\mathcal{S}$, elements in this category can be processed in the same two ways. The difference being that a symbolic evaluation of these axiomatic descriptions is performed when test cases are generated.

3.5 All other axiomatic descriptions ($\mathcal{O}$)

Any axiomatic description not falling within any of the previous categories, belongs to this category. For instance, *checksum* in Sect. 2. An element in this category can be processed in two ways:

1. Users can provide a constant value for it. This value will be used to generate test cases for all the test specification where the axiomatic description appears. The value must help satisfy the predicate part of all axiomatic descriptions in which it appears.
2. Alternatively, Fastest can choose any value for it. Although this way of treating these axiomatic descriptions may increase the degree of automation, it can severely complicate the generation of test cases because some test specifications may become unsatisfiable, when they may not for other values. Furthermore, without any further information Fastest may choose an odd value with respect to the implementation that is going to be tested.

4 Testing Tactics for Quantifications

As we have said in Sect. 2.2, we have decided to approach the generation of test cases when quantifications are used in operations, by defining some testing tactics specially tailored to deal with such predicates. So far, the TTF had treated quantifications as atomic predicates making it very difficult, or even impossible, to generate test cases to exercise the corresponding implementation sentences—usually loops. Hence, in the following sections we introduce these new testing tactics for quantified formulas. These testing tactics can be applied only when: (i) the quantified formula includes predicates depending only on input or before-state variables, and (ii) the sets over which the bound variables ranges, depend on the same kind of variables. These restrictions are reasonable since the whole goal of the TTF is to produce a partition of the input space of the operation, which is defined by all the input and before-state variables.

4.1 Weak Existential Quantifier (WEQ)

Conceptually, this testing tactic transforms a quantification over a potentially infinite set into a quantification over a finite set. Since an existential quantification over a finite set is equivalent to a disjunction, then WEQ first transforms the existential quantification into a disjunction. Then it writes the disjunction into DNF and finally it generates as many test specifications as terms the DNF has plus one more characterized by the negation of the other predicates. In order to apply WEQ the user has to indicate the quantified formula and the maximum number of elements to be considered for each bound variable.

The example depicted in Fig. 1 helps to understand how WEQ works. Assume $M : \mathbb{P}\,\mathbb{N}$, $H : \mathrm{seq}\,\mathbb{Z}$ and $w, v : \mathbb{Z}$ are four input or before-state variables. WEQ_1 and WEQ_2 say that a test case must be generated when $x = x_1$ and $y = y_1$; WEQ_3 and WEQ_4 say the same but with $x = x_2$; and WEQ_5 says there may be other test cases to derive from the formula. x_1, x_2 and y_1 are new identifiers that must instantiated when test cases are generated. Note that, in general, no satisfiability algorithm will be able to automatically generate an abstract test case for all test specifications like WEQ_5, due to the presence of the quantification over potentially infinite sets.

Original predicate	$\exists\, x : M;\ y : H \bullet x > w \wedge (y \neq \langle\rangle \Rightarrow head\ y > v)$
Maximums	$x \leftarrow 2, y \leftarrow 1$
First transformation	$\{x_1, x_2\} \subseteq M \wedge \{y_1\} \subseteq H$ $((x_1 > w \wedge (y_1 \neq \langle\rangle \Rightarrow head\ y_1 > v))$ $\vee\ (x_2 > w \wedge (y_1 \neq \langle\rangle \Rightarrow head\ y_1 > v)))$
Write in DNF	$\{x_1, x_2\} \subseteq M \wedge \{y_1\} \subseteq H$ $(x_1 > w \wedge \neg\ y_1 \neq \langle\rangle$ $\vee\ x_1 > w \wedge head\ y_1 > v$ $\vee\ x_2 > w \wedge \neg\ y_1 \neq \langle\rangle$ $\vee\ x_2 > w \wedge head\ y_1 > v)$
New test specifications	$WEQ_1 \rightarrow x_1 > w \wedge y_1 = \langle\rangle \wedge \{x_1\} \subseteq M \wedge \{y_1\} \subseteq H$ $WEQ_2 \rightarrow x_1 > w \wedge head\ y_1 > v \wedge \{x_1\} \subseteq M \wedge \{y_1\} \subseteq H$ $WEQ_3 \rightarrow x_2 > w \wedge y_1 = \langle\rangle \wedge \{x_2\} \subseteq M \wedge \{y_1\} \subseteq H$ $WEQ_4 \rightarrow x_2 > w \wedge head\ y_1 > v \wedge \{x_2\} \subseteq M \wedge \{y_1\} \subseteq H$ $WEQ_5 \rightarrow (\exists\, x : M;\ y : H \mid x \notin \{x_1, x_2\} \wedge y \notin \{y_1\} \bullet$ $x > w \wedge (y \neq \langle\rangle \Rightarrow head\ y > v))$

Fig. 1. Generating test specifications by applying WEQ.

Likely, an existential quantification will be implemented as an iteration statement that will be abandoned when the first value satisfying its condition is found. Therefore, it is important to test this statement by making it execute zero, one or more iterations. Furthermore, it is important to test the inner clause with different values. WEQ allows all of this by letting users restrict the quantification over a suitable finite set and by transforming the quantified formula into a set of quantified-free formulas. Therefore, once WEQ has been applied, other testing tactics can be applied to the new test specifications. For instance, we can apply Standard Partition [8] over $x > w$ in order to test for positive, zero or negative values of both x and w.

It should be noted that this tactic might not produce a partition of the test specifications—if this is unacceptable, then see the next section.

4.2 Strong Existential Quantifier (SEQ)

This tactic is a stronger form of WEQ since it always generates a partition of the test specifications where it is applied. SEQ conjoins the following predicate to the i^{th} test specification produced by WEQ, except to the last one:

$$\neg\ (\exists\, x_1 : T_1, \ldots, x_n : T_n \mid x_1 \neq v_i^1 \wedge \cdots \wedge x_n \neq v_i^n \bullet P(x_1, \ldots, x_n, x))$$

where $x_1 : T_1, \ldots, x_n : T_n$ are the quantified variables and their types, $v_i^1, \ldots, v_i^n$ are the new variables introduced by WEQ making up the i^{th} combination, and P is the quantified predicate. For instance, in the example shown in Fig. 1, SEQ

would generate the following test specifications:

$$
\begin{aligned}
SEQ_1 \rightarrow\ & x_1 > w \land y_1 = \langle\rangle \land \{x_1\} \subseteq M \land \{y_1\} \subseteq H \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq x_1 \land y \neq y_1 \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > v) \\
SEQ_2 \rightarrow\ & x_1 > w \land head\ y_1 > v \land \{x_1\} \subseteq M \land \{y_1\} \subseteq H \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq x_1 \land y \neq y_1 \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_3 \rightarrow\ & x_2 > w \land y_1 = \langle\rangle \land \{x_2\} \subseteq M \land \{y_1\} \subseteq H \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq x_2 \land y \neq y_1 \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_4 \rightarrow\ & x_2 > w \land head\ y_1 > v \land \{x_2\} \subseteq M \land \{y_1\} \subseteq H \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq x_2 \land y \neq y_1 \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_5 \rightarrow\ & (\exists\, x : M;\ y : H \mid \\
& \qquad x \notin \{x_1, x_2\} \land y \notin \{y_1\} \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x)
\end{aligned}
$$

However, in general, no satisfiability method will be able to automatically generate abstract test cases for any of these test specifications due to the presence of the quantification over an infinite set. Therefore, in spite that WEQ do not produce always a partition, and thus it potentially generates the same test case more than once, it will allow a satisfiability algorithm to find at least some test cases some times. However, SEQ is still valuable since users may provide, manually, test cases satisfying these test specifications, if WEQ is too weak for their needs.

4.3 Universal Quantifications

In order to produce a partition of a universal quantification, we propose a testing tactic, called UQ, that considers different cardinalities for the bound variables. Consider a universal quantification such as:

$$\forall\, x_1 : S_1, \ldots, x_n : S_n \bullet P(x_1, \ldots, x_n, x) \tag{1}$$

where $S_1, \ldots, S_x$ are sets that depend only on input variables and x represents other input variables. Then, users may apply UQ by indicating a limit to the number of elements that will be considered for each x_i. If these limits are $M_1, \ldots, M_n$, then we first generate the following sets for each $i \in 1 \,..\, n$:

$$
\begin{aligned}
& S_i^0 = \emptyset \\
& S_i^1 = \{v_i^1\} \\
& S_i^2 = \{v_i^1, v_i^2\} \\
& \ldots \\
& S_i^{M_i - 1} = \{v_i^1, v_i^2, \ldots, v_i^{M_i - 1}\} \\
& S_i^{M_i} = \{v_i^1, v_i^2, \ldots, v_i^{M_i}\} \\
& S_i^{M_i} \subset S_i^{M_i + 1}
\end{aligned}
$$

where v_i^j are all new identifiers that must be instantiated when test cases are generated. Then, UQ generates the following $(M_1 + 2) \times \cdots \times (M_n + 2)$ test

specifications:

$$\forall\, x_1 : S_1^{j_1}, \ldots, x_n : S_n^{j_n} \bullet P(x_1, \ldots, x_n, \ldots) \tag{2}$$
$$S_1^{j_1} \subseteq S_1 \wedge \cdots \wedge S_n^{j_n} \subseteq S_n$$

where $j^k \in 0 \,..\, M_k + 1$ for all $k \in 1 \,..\, n$. Given that all the $S_i^{j_k}$, for $j^k \in 0 \,..\, M_k$ and for all $i, k \in 1 \,..\, n$, are finite, each quantification is equivalent to a finite conjunction. For uniformity with the other testing tactics, Fastest will also write each formula like (2) in DNF—so all the test specifications are conjunctions of atomic predicates. Although writing (2) in DNF may generate more test specifications we say that UQ generates $(M_1 + 2) \times \cdots \times (M_n + 2)$ test specifications.

As an example of the application of UQ consider the following formula:

$$\forall\, x : M;\ y : H \bullet x > w \wedge (y \neq \langle\rangle \Rightarrow \mathit{head}\, y > v)$$

and assume the user provides the following limits: $x \leftarrow 2$ and $y \leftarrow 1$. First, define $S_1^1 = \{x_1\}$, $S_1^2 = \{x_1, x_2\}$, $S_1^3 = A$, $S_2^1 = \{y_1\}$ and $S_2^2 = B$. Second, for brevity, define $P(x, y, v, w) = x > w \wedge (y \neq \langle\rangle \Rightarrow \mathit{head}\, y > v)$. Therefore, UQ generates the following test specifications:

$$UQ_1 \rightarrow \forall\, x : \emptyset;\ y : \emptyset \bullet P(x, y, v, w)$$
$$UQ_2 \rightarrow (\forall\, x : \{x_1\};\ y : \emptyset \bullet P(x, y, v, w)) \wedge \{x_1\} \subseteq M$$
$$UQ_3 \rightarrow (\forall\, x : \{x_1, x_2\};\ y : \emptyset \bullet P(x, y, v, w)) \wedge \{x_1, x_2\} \subseteq M$$
$$UQ_4 \rightarrow \forall\, x : A;\ y : \emptyset \bullet P(x, y, v, w)) \wedge \{x_1, x_2\} \subset A \wedge A \subseteq M$$
$$UQ_5 \rightarrow (\forall\, x : \emptyset;\ y : \{y_1\} \bullet P(x, y, v, w)) \wedge \{y_1\} \subseteq H$$
$$UQ_6 \rightarrow (\forall\, x : \{x_1\};\ y : \{y_1\} \bullet P(x, y, v, w)) \wedge \{x_1\} \subseteq M \wedge \{y_1\} \subseteq H$$
$$UQ_7 \rightarrow (\forall\, x : \{x_1, x_2\};\ y : \{y_1\} \bullet P(x, y, v, w)) \wedge \{x_1, x_2\} \subseteq M \wedge \{y_1\} \subseteq H$$
$$UQ_8 \rightarrow (\forall\, x : A;\ y : \{y_1\} \bullet P(x, y, v, w)) \wedge \{x_1, x_2\} \subset A \wedge A \subseteq M \wedge \{y_1\} \subseteq H$$
$$UQ_9 \rightarrow (\forall\, x : \emptyset;\ y : B \bullet P(x, y, v, w)) \wedge \{y_1\} \subset B \wedge B \subseteq H$$
$$UQ_{10} \rightarrow (\forall\, x : \{x_1\};\ y : B \bullet P(x, y, v, w)) \wedge \{x_1\} \subseteq M \wedge \{y_1\} \subset B \wedge B \subseteq H$$
$$UQ_{11} \rightarrow (\forall\, x : \{x_1, x_2\};\ y : B \bullet P(x, y, v, w)) \wedge \{x_1, x_2\} \subseteq M \wedge \{y_1\} \subset B \wedge B \subseteq H$$
$$UQ_{12} \rightarrow (\forall\, x : A;\ y : B \bullet P(x, y, v, w)) \wedge \{x_1, x_2\} \subset A \wedge A \subseteq M \wedge \{y_1\} \subset B \wedge B \subseteq H$$

However, Fastest eliminates the quantified formulas by transforming them into conjunctions. For instance, the quantified formula of UQ_7 becomes:

$$\begin{aligned}
&\forall\, x : \{x_1, x_2\};\ y : \{y_1\} \bullet P(x, y) \\
&\equiv \forall\, x : \{x_1, x_2\} \bullet \forall\, y : \{y_1\} \bullet P(x, y) \\
&\equiv \forall\, x : \{x_1, x_2\} \bullet x > w \wedge (y_1 \neq \langle\rangle \Rightarrow \mathit{head}\, y_1 > v) \\
&\equiv (x_1 > w \wedge (y_1 \neq \langle\rangle \Rightarrow \mathit{head}\, y_1 > v)) \wedge (x_2 > w \wedge (y_1 \neq \langle\rangle \Rightarrow \mathit{head}\, y_1 > v))
\end{aligned}$$

In this way, UQ will produce test specifications which ultimately will execute the statement corresponding to the quantified formula, likely an iteration, a different number of times and for different values of the bound variables. As with WEQ and SEQ, UQ can be combined with existing testing tactics in order to produce new test specifications based on the structure of the inner predicated of the quantified formula.

5 Testing Tactics for Set Comprehensions

Any lambda expression can be written as a set comprehension [17, page 58]:

$$(\lambda\, x : X \mid P(x) \bullet f(x)) \equiv \{x : X \mid P(x) \bullet x \mapsto f(x)\}$$

where f is an expression depending on x and possibly on other free variables. Therefore, the ideas presented in this section can also be applied to lambda expressions. In turn, the most general form of a set comprehension in Z is:

$$\{x : X \mid P(x) \bullet \mathit{expr}(x)\}$$

where P is a predicate and *expr* is an expression, both depending on the bound variable and possibly on some free variables. The type of the set comprehension is given by the type of *expr* [17, page 57].

Clearly, the complexity of a set comprehension lies on the complexity of both P and *expr*. As we have said in Sect. 2.3, the idea is to move the complexity of P to the outside of the set comprehension. More precisely:

1. Write P in DNF: $P_1 \vee \cdots \vee P_n$, where each P_i is a conjunction of literals.
2. Rewrite the set comprehension as a set union:

$$\{x : X \mid P(x) \bullet \mathit{expr}(x)\} \equiv \\ \{x : X \mid P_1(x) \bullet \mathit{expr}(x)\} \cup \cdots \cup \{x : X \mid P_n(x) \bullet \mathit{expr}(x)\}$$

3. Rewrite each term of the set union as a set intersection.
4. Apply SP to one or more $\cup$ or $\cap$.

Alternatively, apply SP inside the set comprehension rewriting it as:

$$\{x : X \mid P(x) \wedge (Q_1(x) \vee \cdots \vee Q_n(x)) \bullet \mathit{expr}(x)\}$$

where each Q_i is the i^{th} predicate stipulated by the corresponding standard partition. Then write $P(x) \wedge (Q_1(x) \vee \cdots \vee Q_n(x))$ in DNF and do as above. Yet another alternative is to apply SP to operators appearing in *expr* instead of or apart from P. All these can be combined as is customary in the TTF to further partition previous test specifications. The net effect is a coverage similar to the one delivered by the TTF for other constructions.

6 Concluding Remarks

There are some MBT methods, besides the TTF, for the Z notation [19–24] but none of them approaches axiomatic descriptions, quantified formulas and set comprehensions. Extending any MBT method for the Z notation to deal with these concepts is important because large specifications will include them. Then, all these MBT methods may benefit from our results. Furthermore, MBT methods for other notations such as Alloy, B and VDM may also take advantage of these results since these languages use similar mathematical theories.

Although the paper deals with three somewhat unrelated issues, there is a common, underlying theme: automation. That is, the rules proposed here to process axiomatic descriptions, quantifications and set comprehensions were devised to preserve the degree of automation currently featured by Fastest. Variants of these rules may be proposed but likely they will render the tool less automatic.

Regarding axiomatic descriptions, we conclude that they should be classified according to their intended use before processing them to produce test cases. Although the same language construct is used to define all of them, there are key differences, for example, between declarations such as *root* (Sect. 2.1) and *failed* (Sect. 3.3), making it dangerous to treat them in the same way. The taxonomy presented here may be extended, refined or modified but axiomatic descriptions cannot be treated all in the same way.

It may be argued that recent advances in decision procedures and SMT solving should be used to approach quantified formulas. We have two counterarguments to this point: (a) besides finding a witness satisfying a quantified formula, a partition based on its analysis must be generated in order to get a good coverage of its implementation, and this cannot be done with SMT solvers; and (b) our first results on applying SMT solvers to the TTF show that these tools are not immediately or trivially useful for it [25].

Quantifiers are treated in [26] but for the VDM notation which is based on three value logic and where all sets must be finite—key differences with respect to Z. The rules proposed by Meudec: (i) take into consideration the fact that predicates or expressions can be undefined according to the VDM semantics; (ii) they lead to very long partitions even for basic quantified expressions; and (iii) apparently, these partitions cannot be controlled by the user as ours can. For all these reasons we decided to develop our own tactics to deal with quantifiers.

With regard to future work, we are working on the implementation of the results for set comprehensions and we are further investigating if SMT solvers can be used as a back-end to automate some of the results presented here.

Acknowledgements. This paper is a humble tribute to the memory of David Carrington, one of the creators of the Test Template Framework, who passed away in Australia on 7 January 2011 after a long battle with cancer.

References

1. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering **22**(11) (November 1996) 777–793
2. Stocks, P.: Applying Formal Methods to Software Testing. PhD thesis, Department of Computer Science, University of Queensland (1993)
3. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
4. 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) (2009) 1–76
5. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM (2002) 112–122
6. Legeard, B., Peureux, F., Utting, M.: A Comparison of the BTT and TTF Test-Generation Methods. In: ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, London, UK, Springer-Verlag (2002) 309–329
7. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Softw. Eng. J. **6**(6) (1991) 387–405
8. Cristiá, M., Rodríguez Monetti, P.: Implementing and applying the Stocks-Carrington framework for model-based testing. In Breitman, K., Cavalcanti, A., eds.: ICFEM. Volume 5885 of Lecture Notes in Computer Science., Springer (2009) 167–185
9. Cristiá, M., Albertengo, P., Rodríguez Monetti, P.: Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. In Fiadeiro, J.L., Gnesi, S., eds.: SEFM, IEEE Computer Society (2010) 268–277
10. Cristiá, M.: Fastest tool. Available at: `http://www.fceia.unr.edu.ar/~mcristia`. Last access: November 2011
11. Cristiá, M., Plüss, B.: Generating natural language descriptions of Z test cases. In Kelleher, J.D., Namee, B.M., van der Sluis, I., Belz, A., Gatt, A., Koller, A., eds.: INLG, The Association for Computer Linguistics (2010) 173–177
12. Cristiá, M., Hollmann, D., Albertengo, P., Frydman, C.S., Monetti, P.R.: A language for test case refinement in the Test Template Framework. In Qin, S., Qiu, Z., eds.: ICFEM. Volume 6991 of Lecture Notes in Computer Science., Springer (2011) 601–616
13. Cristiá, M., Santiago, V., Vijaykumar, N.: On comparing and complementing two MBT approaches. In Vargas, F., Cota, E., eds.: LATW, IEEE Computer Society (2010) 1–6
14. Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Rodríguez Monetti, P.: Applying the Test Template Framework to aerospace software. In: Proceedings of the 34th IEEE Annual Software Engineering Workshop, Limerik, Irland, IEEE Computer Society (2011)
15. ECSS: Space Engineering – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization. Technical Report ECSS-E-70-41A, European Space Agency (2003)
16. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, IEEE (2006)

17. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)
18. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Technical report, ORA Canada (1997)
19. Ammann, P., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: Compass'94: 9th Annual Conference on Computer Assurance, Gaithersburg, MD, National Institute of Standards and Technology (1994) 69–80
20. Hall, P.A.V.: Towards testing with respect to formal specification. In: Proc. Second IEE/BCS Conference on Software Engineering. Number 290 in Conference Publication, IEE/BCS (July 1988) 159–163
21. Hierons, R.M., Sadeghipour, S., Singh, H.: Testing a system specified using Statecharts and Z. Information and Software Technology **43**(2) (February 2001) 137–149
22. Hierons, R.M.: Testing from a Z specification. Software Testing, Verification & Reliability **7** (1997) 19–33
23. Hörcher, H.M., Peleska, J.: Using Formal Specifications to Support Software Testing. Software Quality Journal **4** (1995) 309–327
24. Burton, S.: Automated Testing from Z Specifications. Technical report, Department of Computer Science – University of York (2000)
25. Cristiá, M., Frydman, C.: Applying SMT solvers to the Test Template Framework. In Petrenko, A.K., Schlingloff, H., eds.: Proceedings 7th Workshop on Model-Based Testing, Tallinn, Estonia, 25 March 2012. Volume 80 of Electronic Proceedings in Theoretical Computer Science., Open Publishing Association (2012) 28–42
26. Meudec, C.: Automatic generation of software tests from formal specifications. PhD thesis, Queen's University of Belfast, Northern Ireland, UK (1997)