Using {*log*} as a Test Case Generator for Z Specifications

Maximiliano Cristiá[1] and Gianfranco Rossi[2]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar
[2] Università degli Studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it

Abstract. The Test Template Framework (TTF) is a model-based testing method for the Z notation, a formal specification language based on first-order logic and set theory. In the TTF, test cases are generated from test specifications, which are predicates written in Z. In this paper we show how {*log*} can be applied as a test case generator for the TTF. {*log*} is a Constraint Logic Programming language that embodies the fundamental forms of set designation and a number of primitive operations for set management. As such, it can find solutions of first-order logic formulas involving set-theoretic operators. According to our experiments, {*log*} produces promising results as concerns the effectiveness and efficiency in finding solutions (i.e. test cases) out of a number of non-trivial test specifications.

1 Introduction

Model-Based Testing (MBT) attempts to generate test cases to test a program from its specification. These techniques have been proposed for, and applied to, several formal notations such as Z [1], finite state machines and their extensions [2], B [3], algebraic specifications [4], etc. The Test Template Framework (TTF) was first proposed by Stocks and Carrington [1] as a MBT method for the Z notation. Recently, Cristiá and others provided tool support for the TTF by means of Fastest [5–7], and extended it to deal with Z constructs not included in the original presentation [8] and beyond test case generation [9, 10].

Given a Z specification, the TTF takes each Z operation and partitions its input space in a number of so-called *test specifications*. For the purpose of this paper, it does not really matter how these test specifications are generated because the problem we are approaching here starts once they are given. In this context, a test specification is a conjunction of atomic predicates written in the Z notation. That is, a test specification is a conjunction of atomic predicates involving sets as well as binary relations, functions and partial functions, sequences and other mathematical structures as defined in the Z Mathematical Toolkit (ZMT) [11]. Clearly, a test specification can also be seen as the set of elements satisfying the conjunction of atomic predicates.

According to the TTF, a *test case* is an element belonging to a test specification. In other words, a test case is a witness satisfying the predicate that characterizes a test specification. Hence, in order to find a test case for a given test specification it is necessary to find a solution for a Z formula. When Fastest was first implemented (early 2007) a rough, simple satisfiability algorithm was implemented, which proved to be reasonable effective and efficient [5, 7]. However, this algorithm tends to be slow on complex test specifications.

In this paper we analyze the application of a constraint solver for this task. Namely, we consider the constraint solver provided by $\{log\}$ [15–17], a Constraint Logic Programming (CLP) language that embodies the fundamental forms of set designation and a number of primitive operations for set management.

In order to apply $\{log\}$ to the problem of finding test cases from test specifications within the TTF we need, first of all, to define an encoding of (at least a significant portion of) the ZMT into the $\{log\}$'s language. Thus, an original contribution of this paper is to show how $\{log\}$ can be adapted to work with concepts and operators defined in the ZMT and how the latter can be embedded into the former.

Furthermore, we present the results of an empirical assessment of $\{log\}$ used as test case generator for the TTF as concerns the effectiveness and efficiency in finding solutions (i.e. test cases) out of a number of satisfiable test specifications. Given that these test specifications are among the most complex test specifications of eight Z models, some of which are models of real, industrial-strength systems, another original contribution of this paper is to show that $\{log\}$ in particular, and CLP in general, could be used as test case generators for MBT methods.

The encoding of the ZMT into $\{log\}$, plus the results of the empirical assessment presented in this paper and those in [12, 13], may have a non-trivial impact on tools for notations such as VDM, B and even TLA+ and Alloy. In effect, all of these notations are based on similar set theories and, thus, can benefit from the encoding presented here since their users can use $\{log\}$ as a satisfiability solver or a specification animator.

This paper assumes the reader is familiar with the mathematics underlying Z and with general notions of formal software verification. Section 2 introduces the TTF. In Section 3 we show the modifications and extensions introduced in $\{log\}$ to make it more suitable as a test case generator for the TTF. Section 4 presents a translation of a significant portion of the ZMT into $\{log\}$. The results of an empirical assessment are shown in Section 5. Finally, in Sections 6 and 7 we discuss the results shown in this paper and give our conclusions.

2 Test Cases in the TTF

In the TTF, test cases are derived from test specifications. The work presented in this paper starts once test specifications have been generated, making it unnecessary to explain the process to get them. Test specifications are sets satisfying

predicates that depend on input and state variables. In the TTF, both test specifications and test cases are described in Z by means of schemata. For example, the first schema in Fig. 1 corresponds to a test specification borrowed from one of our case studies, which is a Z specification of a real satellite software. In the figure, $BYTE$ is a given type (i.e. uninterpreted sort) and $DTYPE ::= SD|HD|MD$. Observe that although mem does not participate in $TransmitSD_{24}^{SP}$, a test case generator must be able to bind to it a set of 1024 ordered pairs representing a function. The second schema in Figure 1 is a test case (generated by $\{log\}$) for $TransmitSD_{24}^{SP}$. Note how the TTF uses schema inclusion to link test cases with test specifications.

$TransmitSD_{24}^{SP}$

$c, t : DTYPE \to \mathbb{N};\ mem : 1 \,..\, 1024 \to BYTE;\ sdwp : \mathbb{N}$

$c\ SD = 0 \land sdwp < 3 \land 33 \,..\, 160 \neq \varnothing$
$33 + (t\ SD - c\ SD) * 2 \,..\, 33 + (t\ SD - c\ SD + 1) * 2 \neq \varnothing$
$33 \,..\, 160 \cap 33 + (t\ SD - c\ SD) * 2 \,..\, 33 + (t\ SD - c\ SD + 1) * 2 \neq \varnothing$
$\neg\ 33 \,..\, 160 \subseteq 33 + (t\ SD - c\ SD) * 2 \,..\, 33 + (t\ SD - c\ SD + 1) * 2$
$\neg\ 33 + (t\ SD - c\ SD) * 2 \,..\, 33 + (t\ SD - c\ SD + 1) * 2 \subseteq 33 \,..\, 160$
$33 + (t\ SD - c\ SD) * 2 \,..\, 33 + (t\ SD - c\ SD + 1) * 2 \neq 33 \,..\, 160$

$TransmitSD_{24}^{TC}$

$TransmitSD_{24}^{SP}$

$c = \{sd \mapsto 0, hd \mapsto 1, md \mapsto 2\}$
$t = \{sd \mapsto 63, hd \mapsto 0, md \mapsto 1\}$
$sdwp = 0$
$mem = \{1 \mapsto G11084, 2 \mapsto G11116, \ldots \text{and 1022 more elements} \ldots\}$

Fig. 1. Typical test specification and test case in the TTF.

Although this example does not use partial functions nor sequences, these features are heavily used in Z specifications and the TTF works with them. Hence, many of the test specifications used in our empirical assessment include partial functions and sequences, and other set operators as well. Any tool that could be used as test case generator for the TTF should be able to deal with such mathematical objects. Note that the problem here is not the logic structure of the test specification (it is just a conjunction of atomic predicates), but rather the ability to manage efficiently such mathematical objects.

$TransmitSD_{24}^{SP}$ is a satisfiable test specification. However, the TTF tends to generate many unsatisfiable test specifications. Fastest implements a test specification pruning method that proved to be effective, efficient and easily extensible [6, 7]. Hence, we are more concerned in finding a better test case generator rather than a replacement for the pruning method.

3 Improving {*log*} for the TTF

In order to define a translation of the TTF Test Specifications into {*log*} we need primarily to define new predicates that implement fundamental notions of the ZMT that are not directly supported by {*log*}.

The new predicates are defined in a {*log*}'s library specially developed for the TTF. They include predicates for checking whether a set is a binary relation or a partial function, for determining the range of a binary relation or the domain of a sequence, for calculating a function on an argument, and so on.

The following are examples of such predicates.

```
is_rel(R) :- forall(P in R, exists([X,Y], P = [X,Y])).

dom({},{}).
dom({[X,Y]/Rel},Dom) :- dom(Rel,D) & Dom = {X/D} & X nin D.

is_pfun(F) :- forall(P1 in F, forall(P2 in F, nofork(P1,P2))).

nofork([X1,Y1],[X2,Y2]) :- (X1 neq X2 or (X1 = X2 & Y1 = Y2)).
```

`is_rel(R)` is true if `R` is a binary relation, that is a set of pairs of the form `[X,Y]`. `dom(R,D)` is true if `D` is the domain of the relation `R`. `is_pfun(F)` restates the usual ZMT definition of a partial function as a {*log*} predicate: `is_pfun(F)` is true if `F` is a set of ordered pairs where any two of them cannot have the same first component.

The following is a simple goal for the above predicates:

```
R = {[1,5],[2,7]} & is_rel(R) & dom(R,D)
```

and the computed solution for `D` is `D = {1,2}`.

It is important to note that these predicates can be used both to test and to compute values for their arguments. In particular, `is_pfun(F)` can be used not only to test whether `F` is a partial function or not, but it can be used also to build a partial function. This is crucial to make {*log*} a test case generator for the TTF, since most of the variables in most of the test specifications are unbound.

Other fundamental notions of the ZMT are implemented in a similar manner within the {*log*}-TTF library. The availability of general forms of set designation in {*log*} makes this task relatively easy. However, the procedural behavior of this straightforward approach may turn out to be quite unsatisfactory in many cases.

One of the main problems with this solution is the "generality" of the defined predicates. As a matter of fact, the same predicate can be used either to test or to compute values for its arguments, values can be either completely or partially specified and, in the case of set variables, they can be represented either as sets or as intervals. For example, dom(*Rel*, *Dom*) can be used both to compute the domain of a given relation and to compute the relation associated with a given domain This means that, for example, the goal `dom(Rel,int(1,10))` succeeds but it generates through backtracking 10! equivalent solutions—which

are permutations of each other—simply because `int(1,10)` is computed as a set. Similarly, that goal but with a bigger interval, e.g. `int(1,1000)`, takes too much time even to compute the first solution. Though abstractly an interval is just a special case of a set, in practice some operations (e.g., iterating over all its elements) can be performed much more efficiently over intervals than over sets.

To overcome these weaknesses we split the definitions of many of the predicates added to support part of the ZMT into different subcases, which are selected according to the different possible instantiations of their parameters. For example, predicate `dom` has now two different subcases:

```
dom1({},{}).
dom1({[X,Y]/R},D) :- D = {X/S} & X nin S & dom1(R,S).

dom2({[A,Y]},D) :- D = int(A,A).
dom2({[A,Y]/R},D) :-
   D = int(A,B) & A < B & A1 is A + 1 & dom2(R,int(A1,B)).
```

The definition of `dom(Rel,Dom)` is modified accordingly so to allow it to select the proper subcase: `dom1` is selected when `Dom` is either an unbound variable or it is bound to a set; vice versa, `dom2` is selected when `Dom` is bound to an interval (in both cases, `Rel` can be either bound or unbound). With these definitions, the goal shown above, `dom(Rel,int(1,1000))`, terminates in a few milliseconds and it generates one solution only.

Moreover, cases in which the presence of unbound variables may lead to a huge number of different solutions are avoided as much as possible by making use of the *delay* mechanism offered by {*log*}. For example,

```
dsubset(S1,S2) :- delay(subset(S1,S2), nonvar(S1)).
```

defines a version of the predicate `subset` that delays execution of `subset(S1,S2)` while `S1` is unbound. Thus, for example, given the goal `dsubset(S,int(1,100)) & S = {0|R}`, where `S` is an unbound variable, it will be immediately proved to be unsatisfiable since `{0|R}` is trivially proved to be not a subset of `int(1,100)`, whereas the same goal using `subset` would cause 2^{100} different solutions for `S` to be attempted before concluding it is unsatisfiable, leading to unacceptable execution time in practice. Note that, if at the end of the whole computation, a delayed goal is still suspended then it is anyway executed, disregarding its delaying condition.

The second main problem with the straightforward solution presented at the beginning of this section is interval management. In our application, often intervals need to be managed as usual sets, that is, for example, to compute the intersection of two intervals, or the union of two intervals, or the union of an interval and a set, and so on. Moreover, quite often, interval endpoints are not known in advance and are represented as unbound integer variables possibly ranging over some finite domain. For example, we may have to solve goals such as `subset(int(A1,B1),int(A2,B2))`, where some of the interval endpoints `A1`, `A2`, `B1`, `B2` are unbound variables.

Differently from many other solvers over Finite Domains (e.g., CLP(FD)), {*log*} allows intervals to be freely managed as first-class objects of the language

as any other set object [18]. Conversely, as common in constraint solvers dealing with Finite Domains, the current version of {*log*} requires interval endpoints to be completely known when the intervals are processed. This limitation implies that we must force, e.g. by using delay or a suitable subgoal ordering, the management of intervals with unknown endpoints to be postponed until the endpoints possibly get some known value, such as during the final labeling phase. For example, the {*log*} goal `subset(int(10,B),int(1,B)) & B in int(1,1000000)` fails because `B` is unbound when `subset` is processed, while if we delay `subset` until `B` gets a value, the goal succeeds. However, even with the delay, proving that the similar goal `subset(int(0,B),int(1,B))` is unsatisfiable, requires unacceptable execution time simply because the solver has to try all possible values for `B`.

To overcome these limitations we have extended {*log*} so as to allow the set constraint solver to reason about intervals even if they have unknown endpoints. So far the extension is limited to those constraints that are of interest for our specific application, namely equality, membership, inclusion, along with their negative counterparts, and intersection. As an example, the improved version of the constraint `subset`, called `esubset`, is able to deal with the case where both of its arguments are non-empty intervals through the following rule:

```
esubset(S,T) :-
   S=int(I,J) & T=int(K,N) & I =< J & K =< N & I >= K & J =< N.
```

If some endpoints of the involved intervals are unknown, then calling `esubset` simply causes the proper integer constraints over the endpoints to be posted. Thus, for example, the above considered goal but using `esubset` in place of `subset`, i.e. `esubset(int(0,B),int(1,B)) & B in int(1,1000000)`, immediately terminates with a failure, disregarding the order in which subgoals are considered. Note that, the new improved versions of the set constraints take care also of the case in which an interval $\mathsf{int}(a, b)$ represents the empty set, that is whenever $b > a$. For example, the goal `esubset(int(10,B),int(1,B)) & B in int(1,1000000)` succeeds also for all values of `B` which are less than 10 since, in these cases, `int(10,B)` represents the empty set. To account for this, we add another rule to the definition of `esubset`:

```
esubset(S,T) :-
   S=int(I,J) & S = {}.
```

where, in turn, solving `int(I,J) = {}` simply requires to solve the constraint `I > J`.

Similar definitions are provided for all the extended versions of the set constraints dealing with unbounded intervals. This extension, though limited so far to a subset of all the constraints provided by {*log*}, represents a valuable step ahead with respect to what provided by the usual FD constraint solvers, and it is likely to be of general interest also for other applications. For this reason, we plan to extend this improvement to all {*log*} constraints in the near future.

4 Translating TTF Test Specifications into {*log*}

In this section we briefly describe the translation of TTF Test Specifications into {*log*}. The translation is given as a collection of translation rules of the form:

$$\text{rule name}\frac{Z\ \textit{notation}}{\{log\}\ \mathsf{language}}$$

where the text above the line is some Z term and the text below the line is one or more {*log*} formulas, using the new features added to {*log*} as described in the previous section.

Some translation rules are listed in Fig. 2. The rules not shown here can be consulted in [19]. The Z terms are syntactic entities sometimes annotated with their types. For example, in rule seq, X is any type. is_rel, is_pfun, dom, dinters, dsubset and apply are predicates included in the {*log*}'s-TTF library.

$$\mathbb{Z}\frac{\mathbb{Z}}{\mathsf{int}(-10^9, 10^9)} \qquad \text{basic types}\frac{[X]}{\mathsf{set}(X)} \qquad \text{free types}\frac{X ::= c_1 | \ldots | c_n}{X = \{c_1, \ldots, c_n\}}$$

$$\times\frac{x \mapsto y}{[X, Y]} \qquad \text{seq}\frac{s : \operatorname{seq} X}{\mathsf{list}(s)} \qquad \mathbb{P}\,|\,\mathbb{F}\frac{A : (\mathbb{P}\,|\,\mathbb{F})X}{\mathsf{dsubset}(A, X)}$$

$$\leftrightarrow\frac{R : X \leftrightarrow Y}{\mathsf{is\_rel}(R)} \qquad \nrightarrow\frac{f : X \nrightarrow Y}{\mathsf{is\_pfun}(f)} \qquad \rightarrow\frac{f : X \rightarrow Y}{\mathsf{is\_pfun}(f)\ \&\ \mathsf{dom}(f, X)}$$

$$\subseteq\frac{A \subseteq B}{\mathsf{dsubset}(A, B)} \qquad \not\subseteq\frac{\neg\ A \subseteq B}{\mathsf{dnsubset}(A, B)} \qquad \mathsf{apply}\frac{f : X \nrightarrow Y \quad f\ x}{\mathsf{apply}(f, X, Y)}$$

$$\#\frac{A : \mathbb{F}\, X \quad \#A}{\mathsf{size}(A, N)} \qquad \cap\frac{A \cap B}{\mathsf{dinters}(A, B, C)} \qquad \text{dom}\frac{R : X \leftrightarrow Y \quad \operatorname{dom} R}{\mathsf{dom}(R, D)}$$

Fig. 2. Some typical translation rules.

The translation is applied to each test specification in isolation. Each test specification is translated as a {*log*} goal (i.e., a conjunction of either positive or negative atomic predicates). Variable names are translated into the same name but the first letter is written in uppercase. If the translated name conflicts with another identifier then some algorithm to avoid name clashes must be applied.

As part of the preprocessing, all subexpressions of each Z arithmetic expression are replaced for fresh variables and new equalities, binding these fresh variables with the corresponding subexpressions, are added to the test specification. For example, $x * (y + z)$ is translated as M is $Y + Z$ & N is $X * M$. In this way, {*log*} can identify common sub-expressions improving its constraint solving capabilities.

Some rules, such as apply or size, need to introduce fresh variables. In that case, the expression, for instance $f\ x$, is replaced by the new variable. For example, $f\ x > 0$ is translated as $\mathsf{apply}(F, X, Y)$ & $Y > 0$.

Note that the translation rule labeled "basic types" is not really necessary. In effect, given that the elements of basic types have no structure and no properties beyond equality, declaring them in $\{log\}$ is unnecessary because the tool will automatically generate constants as needed. Furthermore, $\{log\}$ will deduce that X is a set if that name participates in a set expression.

This translation works as long as the following hypotheses are satisfied:

Hypothesis 1. The Z specification has been type-checked and all proof obligations concerning domain checks have been discharged [20].

Hypothesis 2. All the test specifications where a partial function is applied outside its domain have been eliminated by running the pruning algorithm implemented in Fastest.

Hypothesis 3. Domain and ranges of binary relations have been normalized.

We believe these hypothesis are reasonable and easy to achieve. If they are not verified before the translation is performed, the solutions returned by $\{log\}$ may turn out to be inconsistent at the Z level. Hypothesis 3 makes it unnecessary to explicit the domain and range of binary relations because $\{log\}$ will generate a binary relation populated by any terms provided they verify the other predicates in the goal (while normalization introduces domain and ranges as predicates). For example, $R : 1 \mathinner{..} 10 \leftrightarrow X$ is normalized as $R : \mathbb{Z} \leftrightarrow X \land \operatorname{dom} R \subseteq 1 \mathinner{..} 10$, which is simply translated as $\mathsf{is\_rel}(R)$ & $\mathsf{dom}(R, D)$ & $\mathsf{dsubset}(D, \mathsf{int}(1, 10))$.

Besides, note that the untyped character of $\{log\}$ does not conflict with Z, at least as a test case generator for the TTF. Consider a Z specification with two basic types, X and Y, and the test specification $[A : \mathbb{P}\, X;\ B : \mathbb{P}\, Y;\ v : X;\ w : Y \mid v \in A \land w \in B]$. When this is translated into $\{log\}$ it becomes: $\mathsf{dsubset}(A, X)$ & $\mathsf{dsubset}(B, Y)$ & V in A & W in B. Since X and Y are unbound variables, part of a possible solution for this goal could be `A = {a}, B = {a}, V = a, W = a`. Although in this paper we are concerned only with the translation from Z to $\{log\}$, we want to emphasize that when a test case returned by $\{log\}$ is translated back to Z the types of the variables at the Z level must be considered. For example, the solution above must be translated as $A = \{aX\} \land B = \{aY\} \land v = aX \land w = aY$, where aX and aY are assumed to be constants of type X and Y, respectively, created during the translation by noting that A and B, at the Z level, have different types.

The following is a typical $\{log\}$ goal, generated by translating the corresponding Z test specification, that the $\{log\}$ solver is able to prove to be satisfiable.

```
set(MDATA) & BIN = {no,yes} & CMODE = {cOF,cON} &
CTYPE = {rM,sC,iDA,sDA,tD,rC,mD,mL,lP} & NAT = int(0, 2147483647) &
Mdp ein NAT & Mep ein NAT & Ped ein NAT & Ctime ein NAT &
Acquiring in BIN & Waiting in BIN & Sending in BIN & Dumping in BIN &
Waitsignal in BIN & Mode in CMODE & Ccmd in CTYPE & Waiting = no &
Ccmd = tD & Sending = Dumping & Dumping = yes & VAR1 is 5*43 &
```

```
INT = int(-2147483648,2147483647) & VAR1 ein INT & Mdp = VAR1 &
list_to_rel(Memp,VAR2) & VAR2 neq {} & VAR3 is Mdp+1 & VAR3 ein INT &
VAR4 is Mdp+43 & VAR4 ein INT & VAR5 = int(VAR3,VAR4) &
dom_list(Memp,VAR6) & einters(VAR5,VAR6,VAR7) & VAR7 neq {} &
ensubset(VAR6,VAR5) & ensubset(VAR5,VAR6) & list(Memp) & list(Memd).
```

where `ein`, `einters`, `eninters` are the extended versions of the corresponding $\{log\}$ set constraints dealing with unbounded intervals, as described in Section 3. Executing such goals by means of the {log} interpreter will produce, in the case goals are satisfiable, a sequence of equalities $X_1 = v_1, \ldots, X_n = v_n$, where $X_1, \ldots, X_n$ are variables and $v_1, \ldots, v_n$ are either variables or constants, along with a (possibly empty) sequence of irreducible $\{log\}$ constraints of some specific forms. This computed answer is then translated back to a Z specification using suitable translation rules from $\{log\}$ to Z (see [19]). The resulting Z formula represents a solution of the original TTF Test Specification, i.e. a test case.

5 Empirical Assessment

In this section we empirically assess $\{log\}$ as a test case generator for the TTF. In particular, we want to know how many test cases and how fast $\{log\}$ can discover.

Since Fastest was first implemented, it has been tested and validated with eleven Z specifications, some of which are formalizations of real requirements. For each of them, a number of test specifications are generated. After eliminating those that are unsatisfiable, Fastest tries to find a test case for the remaining ones. However, it fails to find test cases for 154 out of 475 satisfiable test specifications. In [12, 13], we have chosen 68 of these test specifications for which Fastest fails to evaluate different tools as test case generators for the TTF. Note that we have chosen 68 out of 154 test specifications just because the unchosen specifications belong to the same case study, they are all very similar to each other (in many of them only a variable ranging over an enumerated type changes its value leaving the problematic predicates the same), and similar to some of those included in the experiments. We consider that these test specifications are representative of the problem at hand since, although they are satisfiable, Fastest was unable to solve them, meaning that they are among the most complex.

In order to evaluate $\{log\}$ we make use of this same collection of test specifications. Each specification is translated from Z into the input language of $\{log\}$ by applying the encoding described in Sect. 4

To this purpose we have implemented an automatic translator between Fastest and $\{log\}$. The implementation is a Java program plugged into Fastest and written with ANTLR 4.0 [21]. The translator translates each test specification into a $\{log\}$ goal, calls $\{log\}$ with the goal as input, waits for $\{log\}$'s answer, and translates it back into Z if it returns a solution, otherwise the translator returns "failed". Each $\{log\}$ goal is wrapped inside a `time_out` predicate so infinite runs are avoided. For these experiments a time limit of 10 seconds was used.

The experiments were ran on the following platform: Intel Core^{TM} i3-330M CPU at 2.13GHz with 3 Gb of main memory, running Linux Ubuntu 12.04 (precise) of 32-bit with kernel 3.2.0-30-generic-pae. {*log*} 4.6.17 over SWI-Prolog 5.8.0 for i386 was used during the experiments. The original Z test specifications and their translation into {*log*} can be downloaded from `http://www.fceia.unr.edu.ar/~mcristia/cilc-setlog-ttf.tar.gz`.

N	Case study	R/T	LOZC	State	Oper.	Unsolved
1	Savings accounts (3)	Toy	165	3	6	8
2	Savings accounts (1)	Toy	171	1	5	2
3	Launcher vehicle	Real	139	4	1	8
4	Plavis	Real	608	13	13	29
5	SWPDC	Real	1,238	18	17	12
6	Scheduler	Toy	240	3	10	4
7	Security class	Toy	172	4	7	4
8	Pool of sensors	Toy	46	1	1	1

Table 1. Complexity and size of the case studies.

Table 1 provides some measure of the complexity and size of each case study from which the 68 test specifications were taken (for more information see [7]). **R/T** means whether the Z specification was written from real requirements or not. **LOZC** stands for lines of Z code in LaTeX mark-up. Columns **State** and **Oper** represent the number of state variables and operations, respectively, defined in each specification. **Unsolved** is the number of satisfiable test specifications that Fastest failed to solve in each case study.

Table 2 summarizes the results of this empirical assessment where the meaning of columns is as follows: **Total** is the number of satisfiable test specifications that Fastest was unable to solve (i.e. **Unsolved** in Table 1); **Solved** is the number of test cases found by {*log*}; **Unsolved** is the number of test specifications for which {*log*} failed to find a test case; **Time** is the time spent by Fastest in processing all the test specifications (it includes the time spent on translation, the time spent by {*log*} and the time needed to translate back the results given by {*log*}); and **Percentage** is the percentage of solved test specifications. As can be seen, {*log*} finds in average 84% of the test cases which constitutes an impressive improvement with respect to Fastest and a better performance than other tools with which we have experimented [12, 13]. Besides, the total time required by {*log*} to find the solutions is also very good compared to all of the tools. We have also experimented with larger time limits but this does not seem to increase the number of test cases found but increases the total execution time.

6 Discussion

As mentioned above, we have attempted to use other tools as test case generators for Fastest and the TTF. According to our experiments the best among them is

Case study	Total	Solved	Unsolved	Time	Percentage
Savings accounts (3)	8	8	0	3s	100%
Savings accounts (1)	2	2	0	0.7s	100%
Launcher vehicle	8	8	0	2.5s	100%
Plavis	29	19	10	1m 54s	65%
SWPDC	12	11	1	17s	91%
Scheduler	4	4	0	4s	100%
Security class	4	4	0	1s	100%
Pool of sensors	1	1	0	0.2s	100%
Totals	68	57	11	2m 22s	84%

Table 2. Summary of the empirical assessment.

ProB [14]. ProB is a mainstream tool featuring animation, model-checking and constraint solving capabilities, for the B-Method but also accepting a significant subset of the Z notation. The B notation, in turn, uses a mathematical toolkit similar to the ZMT. Furthermore, at least the constraint solving capabilities of ProB are implemented in Prolog and over CLP(FD). Therefore, a comparison between $\{log\}$ and ProB is a good way of analyzing the relative effectiveness and efficiency of $\{log\}$ for our problem. As can be seen in [13] $\{log\}$ outperforms ProB in the number of test cases found and in the total time spent on that. Moreover, recent additions to $\{log\}$ concerning the management of intervals and some changes in the translation rules produced even better results (i.e Table 2) than those presented in [13].

Then, we think it is worth to analyze the possible causes of the differences shown by the experiments. According to [14], in ProB "sets are represented by Prolog lists" and "any global set of the B machine, ..., will be mapped to a finite domain within SICStus Prolog's CLP(FD) constraint solver". Conversely, $\{log\}$ is based on a well-developed theory of sets and deals with sets and set constraints as first-class entities of the language. Moreover, in order to get better efficiency it combines general set constraint solving with efficient constraint solving over Finite Domains. This combination allows $\{log\}$ to offer various advantages compared to CLP(FD). On the one hand, the presence of very general and flexible set abstractions in $\{log\}$ provides a convenient framework to model problems that are naturally expressed in terms of sets, whereas CLP(FD) may require quite unnatural mappings to integers and sets of integers. On the other hand, the deep combination of the two models, i.e. that of hereditarily finite sets and that of Finite Domains, allows domains in $\{log\}$ to be constructed and manipulated as other sets through general set constraints, rather than having to be completely specified in advance as usual in FD constraint programming. The improvement added to $\{log\}$ for the TTF, which allows intervals to have endpoints with unknown values (see Section 3) is another step ahead with respect to CLP(FD).

The results shown in this paper might indicate that treating sets as first-class objects of a CLP language would be the right choice to further enlarge the class of goals that can be solved in a reasonable time. All this, in turn, might be an indication that sets present fundamental differences with respect to other data structures—such as functions, lists, arrays, etc.—requiring specific theories and algorithms to solve the satisfiability problem of set theory. The results shown in [12] would also indicate that set processing would require a theory such as the one underlying $\{log\}$, and not those underlying SMT solvers. The previous analysis might partially conflict with [22, 23], since in these papers the authors are able to discharge a number of proof obligations generated in B specifications by encoding its mathematical model in some SMT solvers. However, although dual problems, satisfiability is not exactly the same than proof.

Yet another indication reinforcing the previous analysis is the fact that we have observed that $\{log\}$ might not solve some goals because binary relations, partial functions and lists are not treated as first-class entities. For instance, if a goal requires some partial functions to have different cardinalities, but there is no constraint over their elements, $\{log\}$ may iterate over sets of, say, size one trying with different elements, but not different sizes. If there is a large number of elements it would make $\{log\}$ to run for a long time before finding the solution—if it ever terminates. According to the ZMT, lists and (total and partial) functions are all binary relations. Adding specific constraint solving capabilities for binary relations including concepts such as domain and range could make $\{log\}$ to be more effective in dealing with all of them. So far, as shown in sections 3 and 4, binary relations are treated as sets of ordered pairs, i.e. not as first-class objects.

7 Conclusions

We have shown how $\{log\}$ has been improved to use it as a test case generator for the TTF. An empirical assessment suggests that $\{log\}$ performs better than other tools with which we have experimented, in finding more test cases in less time. After these experiments we can say that $\{log\}$ should be considered as a good constraint solver candidate for Fastest and, probably, for other tools of model-based notations such as Z, B, TLA+, Alloy and VDM, given that they are based on similar set theories.

In the near future we plan to write the translator between Z and $\{log\}$ in order to automatize test case generation in Fastest. Also, we will investigate whether or not binary relations (and thus partial functions, sequences, etc.) should be promoted to first-class objects of the CLP language embodied by $\{log\}$, so it improves once again its constraint solving capabilities.

Acknowledgments

This work has been partially supported by the GNCS project “Specifica e verifica di algoritmi tramite strumenti basati sulla teoria degli insiemi”, and by ANPCyT

PICT 2011-1002. We want to thank Joaquín Cuenca and Joaquín Mesuro for their help in implementing the translator between {*log*} and Fastest.

References

1. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Trans. on Software Engineering **22**(11) (Nov. 1996) 777–793
2. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA '02: Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis, ACM (2002) 112–122
3. Legeard, B., Peureux, F., Utting, M.: A Comparison of the BTT and TTF Test-Generation Methods. In: ZB '02: Proc. 2nd Int'l Conf. of B and Z Users on Formal Specification and Development in Z and B, Springer-Verlag (2002) 309–329
4. 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
5. Cristiá, M., Rodríguez Monetti, P.: Implementing and applying the Stocks-Carrington framework for model-based testing. In Breitman, K., Cavalcanti, A., eds.: ICFEM. Vol. 5885 of LNCS., Springer (2009) 167–185
6. 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
7. Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Monetti, P.R.: Tool support for the Test Template Framework. Software Testing, Verification and Reliability pp. n/a–n/a (2012), `http://dx.doi.org/10.1002/stvr.1477`
8. Cristiá, M., Frydman, C.S.: Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions. [24] 280–293
9. 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
10. 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. Vol. 6991 of LNCS., Springer (2011) 601–616
11. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Technical report, ORA Canada (1997)
12. Cristiá, M., Frydman, C.: Applying SMT solvers to the Test Template Framework. In Petrenko, A.K., Schlingloff, H., eds.: Proc. 7th Workshop on Model-Based Testing, Tallinn, Estonia, 25 March 2012. Vol. 80 of Electronic Proc. in Theoretical Computer Science., Open Publishing Association (2012) 28–42
13. Cristiá, M., Rossi, G., Frydman, C.:. {*log*} as a test case generator for the Test Template Framework. In Mario Brevetti, Robert M. Hierons, and Mercedes Merayo, editors, *SEFM*, pages n/a–n/a. n/a, 2013. to appear.
14. Leuschel, M., Butler, M.: ProB: A model checker for B. In Keijiro, A., Gnesi, S., Mandrioli, D., eds.: FME. Vol. 2805 of LNCS., Springer-Verlag (2003) 855–874
15. Dovier, A., Omodeo, E.G., Pontelli, E., Rossi, G.: A language for programming in logic with finite sets. J. Log. Program. **28**(1) (1996) 1–44
16. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. **22**(5) (2000) 861–931
17. Rossi, G.: {*log*}. Available at: `http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html`. Last access:

18. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and clp with sets. In: PPDP, ACM (2003) 219–229
19. Cristiá, M., Rossi, G.: Translation of TTF test specifications into {*log*}. Available at: `http://www.fceia.unr.edu.ar/~mcristia/publicaciones/encoding-ttf-setlog.pdf`. Last access: December 2012
20. Saaltink, M.: The Z/EVES System. In Bowen, J., Hinchey, M., Till, D., eds.: ZUM '97: The Z Formal Specification Notation. (1997) 72–85
21. Parr, T.: The Definitive ANTLR 4 Reference. Oreilly and Associate Series. Pragmatic Programmers, LLC (2013)
22. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT solvers for Rodin. [24] 194–207
23. Mentré, D., Marché, C., Filliâtre, J.C., Asuka, M.: Discharging proof obligations from Atelier B using multiple automated provers. [24] 238–251
24. Derrick, J., Fitzgerald, J.A., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E., eds.: Abstract State Machines, Alloy, B, VDM, and Z - Third Int'l Conf., ABZ 2012, Proceedings. Vol. 7316 of LNCS., Springer (2012)