{*log*} as a Test Case Generator for the Test Template Framework

Maximiliano Cristiá[1], Gianfranco Rossi[2], and Claudia Frydman[3]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar
[2] Università degli studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it
[3] LSIS-CIFASIS, Marseille, France
claudia.frydman@lsis.org

Abstract. {*log*} (pronounced 'setlog') 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. The Test Template Framework (TTF) is a model-based testing method for the Z notation. In the TTF, test cases are generated from test specifications, which are predicates written in Z. In turn, the Z notation is based on first-order logic and set theory. In this paper we show how {*log*} can be applied as a test case generator for the TTF. According to our experiments, {*log*} produces promising results compared to other powerful constraint solvers supporting the Z notation, such as ProB.

1 Seeking a Test Case Generator for the TTF

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. Furthermore, given the advances in the field of tools such as SMT Solvers [12] and Constraint Solvers [13, 14] it is worth to evaluate them as test case generators for the TTF since this is a clear application for them.

In [16] we have analyzed the application of SMT Solvers for this task. Concerning the TTF and the way it works, our results with SMT Solvers were not entirely satisfactory since these tools found just a few test cases. It is important to observe that the ZMT defines some mathematical concepts in a different way with respect to SMT Solvers. For example, in the ZMT the set of functions is included in the set of partial functions, which is included in the type of binary relations, which in turn is the power set of any cross product. SMT Solvers usually do not define the concept of partial functions but only total functions, and in that case they are primitive objects (i.e. they are not defined as sets of ordered pairs). This makes it difficult to use these tools for the TTF.

In this paper we extend our analysis to two Constraint Solvers, namely ProB [17] and $\{log\}$ [18, 19]. $\{log\}$ is a Constraint Logic Programming (CLP) language that embodies the fundamental forms of set designation and a number of primitive operations for set management; and ProB is an animator and model checker, featuring constraint solving capabilities, for the B-Method but also accepting a significant subset of the Z notation. Both ProB and $\{log\}$ natively support sets and set-theoretic operations.

In order to apply these solvers to the problem of finding test cases from test specifications within the TTF it is necessary to define an encoding of (at least a significant portion of) the ZMT into the input languages of the solvers. While the embedding of the ZMT into ProB turns out to be quite natural, the embedding of the ZMT into $\{log\}$ has not been investigated before.

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 $\{log\}$ and ProB used as test case generators for the TTF, in which we compare the effectiveness and efficiency of both systems in finding solutions (i.e. test cases) out of a number of satisfiable test specifications. While both $\{log\}$ and ProB show good performances when compared with Fastest, it seems that the former, with the proposed extensions, can get better results than the latter as regards the specific application taken into account (i.e. it finds more test cases in less time).

The encoding of the ZMT into $\{log\}$, plus the results of the empirical assessment and those presented in [16], 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 either Z or B and with general notions of formal software verification. Sections 2 and 3 introduce the TTF and $\{log\}$, respectively. In Section 4 we show the modifications and extensions introduced in $\{log\}$ to make it more suitable as a test case generator for the TTF. Section 5 presents an encoding of a significant portion of the ZMT into the input language of $\{log\}$. The results of an empirical assessment comparing $\{log\}$ and ProB are shown in Section 6. Finally, in Sections 7 and 8 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.

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 Solving Set Formulas with $\{log\}$

$\{log\}$ [18–20] is a CLP language that extends Prolog with general forms of set data structures and basic set-theoretic operations in the form of primitive con-

$TransmitSD_{24}^{SP}$

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

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

$TransmitSD_{24}^{TC}$

$TransmitSD_{24}^{SP}$

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

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

straints. Sets are primarily designated by *set terms*, that is, terms of one of the forms: $\{\}$, whose interpretation is the empty set, or $\{t_1, \ldots, t_n \mid s\}$, where s is a set term, whose interpretation is the set $\{t_1\} \cup \{t_2\} \cup \cdots \cup \{t_n\} \cup s$. The kind of sets that can be constructed in $\{log\}$ are the so-called *hereditarily finite sets*, that is finitely nested sets that are finite at each level of nesting. For example,

`{1,2,3}`, `{X,{{},{a}},{{{b}}}}`, and `{X|S}`

are all admissible set terms. Note that properties of the set constructor, namely permutativity and right absorption, allow the order and repetition of elements in the set term to be immaterial. Thus, for example, the set terms `{1,1,2}`, `{2,1}`, and `{1,2}` all denote the same set composed of two elements, 1 and 2. Note that similarly to Prolog's lists, a set $\{t_1, \ldots, t_n \mid s\}$ can be partially specified, in that either some of its elements $t_1, \ldots, t_n$ or the remaining part s can contain unbound variables (hence "unknowns").

Sets can be also denoted intensionally by set formers of the form $\{\, X : \mathsf{exists}([\, Y_1, \ldots, Y_n], G)\,\}$, where G is a $\{log\}$-goal (see below) and X, $Y_1, \ldots, Y_n$ are variables occurring in G. The logical meaning of the intensional definition of a set s is $\forall\, X(X \in s \leftrightarrow \exists\, X, Y_1, \ldots, Y_n(G))$. The procedural treatment of an intensional definition in $\{log\}$, however, is based on set grouping (see, e.g., [21]): collect in the set s all instances of X satisfying G for some instantiation of $Y_1, \ldots, Y_n$. Thus intensional set formers are always replaced by the corresponding extensional sets. Obviously, this limits the applicability of intensional set formers to cases in which the denoted set is finite and relatively "small".

Finally, sets can be denoted by *interval terms*, that is terms of the form $\mathsf{int}(a,b)$, where a and b are integer constants, whose interpretation is the inte-

ger interval $[a, b]$. Differently from intensional sets, however, intervals are not converted to the corresponding extensional sets; rather, constraints dealing with intervals directly work on the endpoints of the intervals.

Basic set-theoretic operations are provided in $\{log\}$ as predefined predicates, and dealt with as constraints. For example, the predicates in and nin are used to represent membership and not membership, respectively, the predicate subset represents set inclusion (i.e., subset(r, s) holds iff $r \subseteq s$ holds), while inters represents the intersection relations (i.e., inters(r, s, t) holds iff $t = r \cap s$). Basically, a $\{log\}$-*constraint* is a conjunction of such atomic predicates. For example,

```
1 in R & 1 nin S & inters(R,S,T) & T = {X}
```

where R, S, T and X are variables, is an admissible $\{log\}$-constraint, whose interpretation states that set T is the intersection between sets R and S, R must contain 1 and S must not, and T must be a singleton set.

The original collection of set-based primitive constraints has been extended in [22] to include simple integer arithmetic constraints over Finite Domains as provided by CLP(FD) systems (cf. e.g. [15]). Thus the set of predefined predicates in $\{log\}$ includes predicates to represent the usual comparison relations, such as $<$, $>$, $=<$, etc., whereas the set of function symbols includes integer constants and symbols to represent the standard arithmetic operations, such as $+, -, *,$ div, etc. Accordingly, a $\{log\}$-constraint can be formed by set predicates as well as by integer comparison predicates. For example,

```
inters(R,S,T) & size(T,N) & N =< 2
```

states that the cardinality of $R \cap S$ must be not greater than 2.

The $\{log\}$-interpreter contains a *constraint solver* that is able to check satisfiability of $\{log\}$-constraints with respect to the underlying set and integer arithmetic theories. Moreover, when a constraint c holds, the constraint solver is able to compute, one after the other, all its solutions (i.e., all viable assignments of values to variables occurring in c). In particular, automatic labeling is called at the end of the computation to force the assignment of values from their domains to all integer variables occurring in the constraint, leading to a chronological backtracking search of the solution space. For example, the constraint:

```
X in int(1,5) & Y in int(4,10) & inters({X},{Y},R) & X >= Y
```

is proved to be satisfiable and the following three solutions are computed:

```
X = 4, Y = 4, R = {4};   X = 5, Y = 5, R = {5};   X = 5, Y = 4, R = {}.
```

Possibly remaining irreducible constraints are also returned as part of the computed answer for a given constraint. For example, solving the constraint `inters({1},{Y},R)` will return the two following answers:

```
R = {1}, Y = 1;   R = {}, Y neq 1.
```

Clauses, goals, and programs in $\{log\}$ are defined as usual in CLP. In particular, a $\{log\}$-*goal* is a formula of the form B_1 & B_2 & ... & B_k, where $B_1, \ldots, B_k$ are either user-defined atomic predicates, or atomic $\{log\}$-constraints, or disjunctions of either user-defined or predefined predicates, or Restricted Universal Quantifiers (RUQs). Disjunctions have the form G_1 or G_2, where G_1 and G_2

are $\{log\}$-goals, and are dealt with through nondeterminism: if G_1 fails then the computation backtracks, and G_2 is considered instead. RUQs are atoms of the form $\mathsf{forall}(X \ \mathsf{in} \ s, \mathsf{exists}([Y_1, \ldots, Y_n], G))$, where s denotes a set and G is a $\{log\}$-goal containing $X, Y_1, \ldots, Y_n$. The logical meaning of this atom is $\forall X(X \in s \rightarrow \exists Y_1, \ldots, Y_n(G))$, that is G represents a property that all elements of s are required to satisfy. When s has a known value, the RUQ can be used to iterate over s, whereas, when s is unbound, the RUQ allows s to be nondeterministically bound to all sets satisfying the property G. For example, the goal `forall(X in S,X in {1,2,3})` will bound `S` to all possible subsets of the set `{1,2,3}`. The following is an example of a $\{log\}$ program:

```
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.
```

This program defines two predicates, `is_rel` and `dom`. `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`. The following is a goal for the above program:

```
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 `is_rel(R)` can be used both to test and to compute `R`; similarly, `dom(R,D)` can be used both to compute `D` from `R`, and to compute `R` from `D`, or simply to test whether the relation represented by `dom` holds or not.

$\{log\}$ is fully implemented in Prolog and can be downloaded from [20]. It can be used both as a stand-alone interactive interpreter and as a Prolog library within any Prolog program.

4 Improving $\{log\}$ for the TTF

In order to use $\{log\}$ as a test case generator for the TTF we need to shown how (at least) a significant portion of the ZMT can be embedded into $\{log\}$'s language. This requires primarily the definition of 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.

An example of one of such predicates is the predicate `is_rel` shown in Section 3: `is_rel(R)` is true if the set `R` is a binary relation.

As another example, the following clauses restate the usual ZMT definition of a partial function as a $\{log\}$ predicate: `is_pfun(F)` is true if `F` is set of ordered pairs where any two of them cannot have the same first component:

```
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)).
```

Note that if the the second disjunct in `nofork` is omitted then `is_pfun(F)` can only be used to test if `F` is a partial function or not, but it cannot be used to build a partial function. In that case, calling `is_pfun(F)` with `F` unbound, will return only the solution `F = {}` and nothing else. Therefore, the second disjunct in `nofork` is crucial to make {*log*} a test case generator for the TTF.

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 that often intervals need to be processed even if their endpoints are not precisely known yet. For example, we would like to solve a goal such as `subset(int(A1,B1),int(A2,B2))`, where some of the interval endpoints `A1`, `A2`, `B1`, `B2` are unbound variables. Unfortunately, the current version of {*log*} does not allow this kind of generality in interval management. As is common in constraint solvers dealing with Finite Domains (e.g., CLP(FD)), interval endpoints in {*log*} must be integer constants. However, differently from many other solvers, {*log*} allows intervals to be managed as first-class objects of the language, being intervals just a special case of sets. For example, we can compute the intersection of two intervals, or the union of two intervals, or the union of an interval and a set, and so on. The endpoints of the involved intervals, however, must be known.

To overcome these limitations, at least for those cases that are of interest for our specific application, we define new versions of the primitive constraints dealing with intervals whose endpoints can be unknown. For example, the improved version of constraint `subset`, called `isubset`, deals efficiently with the case where both of its arguments are intervals, through the following predicate:

```
intint_subset(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 `isubset` simply causes the proper integer constraints over the endpoints to be posted. Note that we require that in an interval $\mathsf{int}(a, b)$, b is always greater or equal than a. We exclude the possibility that $\mathsf{int}(a, b)$ with $b > a$ is interpreted as the empty set, which conversely was previously allowed in {*log*}. In fact, giving this possibility would cause the empty set to have an infinite number of different denotations, which may turn out to be very unpractical when interval endpoints are allowed to be unknown and solutions for them must be computed explicitly. Finally, note that the delayed version of the {*log*} predicates for the TTF are modified so as to use these improved versions in place of the usual set constraints (e.g., `dsubset` uses `isubset` in place of `subset`).

The improved versions of the set constraints have been added to {*log*} as user-defined predicates but they will possibly be moved to the interpreter level once a general algorithm for all these special cases is found. As a matter of fact, allowing partially specified sets and intervals with unknown endpoints to be used freely in set constraints requires non-trivial problems to be solved. For instance, even the simple equation $\mathsf{int}(A, B) = \{1, Y, 5, X, 4 \,|\, R\}$, where X, Y, A, B, and R are unbound variables, has no obvious solution. Therefore such kind of generalizations are left for future work.

5 Embedding the ZMT into {*log*}

In this section we present an embedding of the ZMT into {*log*}, in which we extensively exploit the new features added to {*log*} introduced in the previous section. The embedding rules are given as follows:

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

where the text above the line is some Z term and the text below the line is one or more {*log*} formulas. Some embedding rules are listed in Fig. 2. The rules not shown here can be consulted in [23]. The Z terms are syntactic entities sometimes annotated with their types. For example, in rule seq, X is any type.

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

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

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

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

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

Fig. 2. Some typical embedding rules.

The embedding 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. It should be noted that the constants declared in rule "free types" must all start with a lowercase letter because otherwise {*log*} will regard them as variables. Note that ordered pairs are embedded as Prolog lists of length two. 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 embedded as $\mathsf{apply}(F, X, Y)\ \texttt{\&}\ Y > 0$. is_rel, is_pfun, dom, dinters, dsubset and apply are predicates included in the {*log*}'s-TTF library.

There are some embedding rules not shown in the figure. Lower-case variables declared in Z are embedded with a name starting with upper-case, since

otherwise $\{log\}$ takes them as constants. Given a Z arithmetic expression, each sub-expression is given a name by introducing a new variable which is later used to form the full expression. For example, $x * (y + z)$ is embedded as M is $Y + Z$ & N is $X * M$. In this way, $\{log\}$ can identify common sub-expressions improving its constraint solving capabilities.

This encoding 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 [24].

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 $R : \mathbb{Z} \leftrightarrow X \land \mathrm{dom}\, R \subseteq 1 \mathinner{..} 10$, which is simply translated as is_rel(R) & dom(R, D) & dsubset(D, 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: dsubset(A, X) & 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.

6 Empirical Assessment

In this section we empirically assess $\{log\}$ as a test case generator for the TTF. In order to evaluate its effectiveness and efficiency we compare it with ProB, which is a mainstream tool with constraint solving capabilities for the B notation (which in turn uses a mathematical toolkit similar to the ZMT).

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 [16], we have chosen 68 of these test specifications for which Fastest fails to evaluate different tools as test case generators for the TTF[4]. 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*} and compare it with ProB we make use of this same collection of test specifications. Each specification is translated from Z into the input languages of {*log*} by applying the encoding described in Sect. 5, and to ProB (in this case the encoding is straightforward requiring only a syntactic translation). So far, the translation is done "by hand", since we consider that implementing an automatic translator before having some evidence of what tool is the best test case generator for Fastest could have been a waste of time. At the same time, the manual translation can be as unreliable as an unverified program implementing the translation. To minimize errors in the translations, however, all the test specifications were manually verified by two different persons besides who wrote them. The Z test specifications and their corresponding translations will become test cases for the automatic translator that has been started after the assessment was completed.

These experiments were ran on the following platform: Intel Core^{TM} i5-2410M CPU at 2.30GHz with 4 Gb of main memory, running Linux Ubuntu 12.04 (precise) of 32-bit with kernel 3.2.0-30-generic-pae. {*log*} 4.6.16 over SWI-Prolog 5.8.0 for i386 and ProB 1.3.5-beta14 over SICStus Prolog 4.2.0 (x86-linux-glibc2.7) were used during the experiments. The original Z test specifications and their translation into {*log*} and ProB can be downloaded from `http://www.fceia.unr.edu.ar/~mcristia/setlog-ttf.tar.gz`. The translation of each test specification is saved in a file ready to be loaded into the corresponding tool. Scripts to run the experiments are also provided. The results can be analyzed with simple `grep` commands.

We ran two experiments for each tool differing in the timeouts set to let the tools to find a solution for each test specification (otherwise they may run forever in some goals). The two timeouts are 1 second and 1 minute. Hence, both tools can return two possible answers: a) the solution for the goal; or b) some error condition like timeout or an indication that the goal cannot be solved due to some limitation of the tool.

The intention of Table 1 is to provide 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

[4] We have chosen 68 test specifications out of 154 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.

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.

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. As can be seen, the table is divided in two parts. The first one shows the figures for ProB, and the second those for {*log*}. Each part, in turn, is divided into the two experiments ran for each tool. For each experiment the number of solved goals (**Sol**) and unsolved goals (**Uns**) of each case study, are shown. The last row of the table shows the time spent by each tool in processing the 68 goals for each experiment.

As can be seen, these experiments show that {*log*} outperforms ProB in the number of solved goals and in the time spent in doing that. In the 1 second experiment, {*log*} solves 52 goals in 29 seconds while ProB solves 40 in 1 minute, that is a 30% increase in effectiveness and a 50% increase in efficiency. Despite of what Table 2 may suggest, {*log*} does not solve all the goals that ProB does. Indeed, in case studies 4 and 5 both tools discover the same number of test cases but each tool solves goals that the other does not. Combining all the goals solved by both tools, in the 1 minute experiment we get a total of 58 goals solved. This suggests that combining both tools can be beneficial for Fastest and that there are more improvements to add to {*log*}. Note that the tools differ the most in the 1 second experiments, where {*log*} solves 52 goals and ProB 40. This might suggest that {*log*} implements rules that initially narrow the search space better than ProB. The fact that sets in ProB are implemented as Prolog's lists whereas in {*log*} they are first-class objects, might also have a non-negligible impact.

7 Discussion

According to [17], 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*}

N	ProB				{*log*}			
	1 s		1 m		1 s		1 m	
	Sol	Uns	Sol	Uns	Sol	Uns	Sol	Uns
1	7	1	7	1	8		8	
2	1	1	1	1	2		2	
3	8		8		8		8	
4	17	12	17	12	17	12	17	12
5		12	10	2	10	2	10	2
6	2	2	2	2	2	2	4	
7	4		4		4		4	
8	1		1		1		1	
Totals	40	28	50	18	52	16	54	14
Time	1 m 0 s		19 m 40 s		0 m 29 s		13 m 43 s	

Table 2. Summary of the empirical results.

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 4) 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 [16] 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 [25, 26], 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 4 and 5, binary relations are treated as sets of ordered pairs, i.e. not as first-class objects.

8 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 $\{log\}$ would perform better than ProB, 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 "Specifiche insiemistiche eseguibili e loro verifica formale", and by ANPCyT PICT 2011-1002.

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. [27] 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. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (November 2006) 937–977
13. Apt, K.R., Fages, F., Rossi, F., Szeredi, P., Vancza J., eds: Recent Advances in Constraints. Vol. 3010 of LNCS., Springer (2004)
14. Stuckey, P.J., Becket, R., Fischer, J.: Philosophy of the minizinc challenge. Constraints **15**(3) (July 2010) 307–316
15. Schulte, C., Carlsson, M.: Finite Domain Constraint Programming Systems. In Rossi, F., van Beek, P., and Walsh, T., eds: Handbook of Constraint Programming. Elsevier (2006) 493–524
16. 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
17. 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
18. 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
19. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. **22**(5) (2000) 861–931
20. Rossi, G.: {*log*}. Available at: `http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html`. Last access:
21. Dovier, A., Pontelli E., and Rossi, G.: Intensional Sets in CLP. In Palamidessi, C., ed., Logic Programming, 19th Int'l Conf., ICLP2003, Vol. 2916 of LNCS, Springer-Verlag (2003), 284-299
22. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and clp with sets. In: PPDP, ACM (2003) 219–229
23. 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
24. Saaltink, M.: The Z/EVES System. In Bowen, J., Hinchey, M., Till, D., eds.: ZUM '97: The Z Formal Specification Notation. (1997) 72–85
25. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT solvers for Rodin. [27] 194–207
26. Mentré, D., Marché, C., Filliâtre, J.C., Asuka, M.: Discharging proof obligations from Atelier B using multiple automated provers. [27] 238–251
27. 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)