Quaderni del Dipartimento di Matematica e Informatica
Università degli Studi di Parma

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

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

12 dicembre 2012 n. 508

Il presente lavoro è stato finanziato in parte dal progetto GNCS "Specifiche insiemistiche eseguibili e loro verifica formale" e da ANPCyT PICT 2011-1002.

[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

{*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 (CLP) 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 Zermelo-Fraenkel set theory 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 the Zermelo-Fraenkel 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*} performs better than ProB for this application.

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 [5,6,7], 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 over the Zermelo-Fraenkel set theory augmented with 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 characterises 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 tend 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] it is worth to evaluate them as test case generators for the TTF since this is a clear application for them. In [14] two of us analyzed the application of SMT Solvers for this task; and in an unpublished paper [15] we extended these results to two Constraint Solvers, namely ProB [22] and $\{log\}$ [16,17]. $\{log\}$ is a Constraint Logic Programming language that embodies the fundamental forms of set designation and a number of primitive operations for set management; and ProB is animator and model checker, featuring constraint solving capabilities, for the B-Method but also accepting a significant subset of the Z notation. After the empirical assessment carried on during these previous works we concluded that Constraint Solvers perform better than SMT Solvers for this problem, i.e. they find more test cases in less time. Also, we observed that $\{log\}$ could be improved to perform much better.

It is important to mention that, so far, no SMT Solver directly supports the Zermelo-Fraenkel set theory. On the other hand, ProB and $\{log\}$ both natively support that theory. It is also very important to observe that the ZMT defines some mathematical concepts in a different way with respect to such tools as Theorem Provers and 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. More formally, $X \to Y \subset X \pfun Y \subset X \rel Y = \mathbb{P}(X \times Y)$. SMT Solvers and Theorem Provers 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 report on the modifications and extensions introduced in $\{log\}$ to improve it as a test case generator for the TTF, and we present the results of an empirical assessment which clearly shows that now $\{log\}$ performs better than ProB's constraint solver for this task (i.e. it finds more test cases in less time), and consequently it performs better than three SMT Solvers [14,15].

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 a shallow embedding of a significant portion of the ZMT into the input language of $\{log\}$. The results of a new round of an empirical assessment are shown in Section 6 confirming our intuitions. Finally, in Sections 7 and 8 we discuss the results shown in this paper and give our conclusions.

2 Test Cases in the TTF

As we have explained in the introduction, in the TTF, test cases are derived from test specifications. 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 schema. 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.

$TransmitSD_{24}^{SP}$

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

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

$TransmitSD_{24}^{TC}$

$$TransmitSD_{24}^{SP}$$

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

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

Although this example does not use partial functions nor sequences, they are heavily used in Z and B 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 must support this mathematics. Note that the logic structure of the test specification is not the problem (it is just a conjunction of atomic predicates), but its mathematics.

$TransmitSD_{24}^{SP}$ 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*} [16,17,18] is a Constraint Logic Programming language that extends Prolog with general forms of set data structures and basic set-theoretical operations in the form of primitive constraints. 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 some of its elements $t_1, \ldots, t_n$ and/or the rest 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., [19]): 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 integer 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-theoretical operations of Zermelo-Fraenkel set theory are provided in {*log*} as predefined predicates, and dealt with as constraints. For example, predicates in and nin are used to represent membership and not membership, respectively, predicate subset represents set inclusion (i.e., $\mathsf{subset}(r, s)$ holds iff $r \subseteq s$ holds), while inters represents the intersection relations (i.e., $\mathsf{inters}(r, s, t)$ 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 the a singleton set.

The original collection of set-based primitive constraints has been extended in [20] to include simple integer arithmetic constraints over Finite Domains as provided by CLP(FD) systems (cf. e.g. [13]). 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* which 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 space of solutions. For example, the constraint

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

is proved to be true 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 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. 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. Restricted Universal Quantifiers (RUQ) are atoms of the form forall(X in s, 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. For example, solving the goal `is_rel(R)`, with `R` unbound, we get the following solutions for `R`:

`R = {}`; `R = {[X_1,X_2]}`; `R = {[X_1,X_2],[X_3,X_4]}, X_3 neq X_1`

and so on.

{*log*} is fully implemented in Prolog. It can be used both as a stand-alone interactive interpreter and as a Prolog library within any Prolog program. To use {*log*} within Prolog one must call the Prolog predicate setlog(G, C), where G is any {*log*} goal and C is the (possibly empty) list of computed {*log*}-constraints.

4 Improving {*log*} for the TTF

In [15] we have shown that {*log*} can be used as the engine to find test cases from TTF's test specifications, and we have empirically compared it with other Constraint and SMT solvers. In order to use {*log*} for the TTF we have shown how a significant portion of the Z notation can be embedded into the {*log*}'s language. This requires primarily the definition of a collection of new predicates that implement fundamental notions of the ZMT, such as finding if a set is a binary relation or a partial function, finding the range of a binary relation or the domain of a sequence, calculating a function on an argument, etc. For example, checking if a set R is a binary relation is implemented by a call to the predicate `is_rel` shown in Section 3, while finding if a set f is a (total) function, from some set X to some other set, can be implemented by the conjunction

```
is_pfun(f) & dom(f,X)
```

where `dom` is defined as in Section 3, and `is_pfun` can be defined as follows:

```
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_pfun` restates as $\{log\}$ clauses the usual ZMT definition of a partial function as a set of ordered pairs where any two of them cannot have the same first component. 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.

This embedding has been shown in [15] to be feasible and quite "good"—in terms of how many test specifications can be satisfied and in how much time—, though not the best one. In the same paper, however, we observed that $\{log\}$ could be improved to perform much better. In this section we briefly report on some of the enhancements that have been introduced to improve results for the proposed embedding of Z into $\{log\}$. In Sect. 6, we will empirically assess these results and we will compare them with those obtained using ProB.

One of the main problems with the solution presented in [15] is the "generality" of predicates used for the embedding of Z into $\{log\}$. 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, $\mathsf{dom}(Rel, Dom)$ can be used both to compute the domain of a given relation (in which case *Dom* is bound and *Rel* is not) and to compute the relation associated with a given domain (in which case *Dom* is unbound and *Rel* is bound to a known value, that in turn may be either a set or an interval). Though the definitions given in [15] allow this kind of "generality", their procedural behavior may turn out to be quite unsatisfactory in many cases. 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 dealt with as a set. For the same reasons, the same goal 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 efficient over intervals than over sets.

To overcome these weaknesses we split the definitions of many of the predicates reported in [15] into different subcases, which are selected according to the different possible instantiations of their parameters. For example, predicate `dom` has now two different subcases:

```
dom2({},{}).
dom2({[X,Y]/R},D) :-
   D = {X/S} & X nin S & dom2(R,S).
```

```
dom3({[A,Y]},D) :-
   D = int(A,A).
dom3({[A,Y]/R},D) :-
   D = int(A,B) & A < B & A1 is A + 1 & dom3(R,int(A1,B)).
```

The definition of `dom(Rel,Dom)` is modified accordingly so to allow it to select the proper subcase: `dom2` is selected when `Dom` is either an unbound variable or it is bound to a set; vice versa, `dom3` 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 new 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 but 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.

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 solution presented in [15] 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.

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 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:

```
ii_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 be possibly 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 not trivial problems to be solved. For instance, even the simple equation $\mathsf{int}(A, B) = \{1, Y, 5, X, 4 \mid R\}$, where X, Y, A, B, and R are unbound variables, has no obvious solution. Therefore such kind of generalizations at the interpreter level are left for future work.

5 A Shallow Embedding of the ZMT in $\{log\}$

The embedding of the ZMT into $\{log\}$ presented in this paper is similar, but not equal, to the one described in [15]. The embedding rules are 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 Z features are omitted because they are outside the scope of this paper; and some rules are not given because they can be easily deduced from the others—for example, a rule for set intersection is given but not for set union. The main embedding rules are listed in Fig. 2.

The embedding rule labelled "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 identifiers 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 $\mathsf{apply}(F, X, Y)$ `&` $Y > 0$. is_rel, is_pfun, dom, dinters, dsubset and apply are predicates included in $\{log\}$'s library.

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

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

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

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

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

Fig. 2. Some typical embedding rules.

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 embedding works as long as the following hypothesis are satisfied:

Hypothesis 1. The Z specification has been type-checked and all proof obligations concerning domain checks might be discharged [21].

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 embedding is ran, 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).

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 $[A : \mathbb{P}\, X;\ B : \mathbb{P}\, Y;\ v : X;\ w : Y | v \in A \land w \in B]$. 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`. If this solution is translated back to Z not considering the types at the Z level, the result

will not be type correct. However, if the translation appropriately qualifies constants, the result will be correct. 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—they can be declared in an axiomatic definition.

6 Empirical Assessment

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. We have chosen 68 of these 154 from eight case studies to evaluate different tools as test case generators for the TTF [14,15]. 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 [15] we compared the effectiveness and efficiency of {*log*} and ProB in finding solutions for these 68 satisfiable test specifications. In order to do that, we manually translated these 68 test specifications from Z into the input languages of {*log*} and ProB by applying shallow embeddings like the one described in Sect. 5. According to those experiments ProB and {*log*}, in that order, were the best among five tools (i.e. they found more test cases in less time). Therefore, after extending {*log*} we reran the same experiments to measure the relative improvement gained with the modifications. The comparison with ProB is important for us given that this tool is being developed since many years and there is a spin-off company applying it in industrial-strength projects.

These new experiments where 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 analysed with simple `grep` commands.

As in [15], 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.

Each {*log*} goal was executed with the following command:

```
prolog -G256m < goalFile
```

where `G256m` makes Prolog to use 256 Mb of memory and `goalFile` is a file with the following structure:

```
time_out(setlog(goal, _CONSTR), T, _RET).
```

where `time_out` is a Prolog function to run a goal for a specified amount of time, `setlog` calls the {*log*} environment from Prolog, `goal` is one of the test specifications and `T` is one of the two timeouts. In turn, each ProB goal was executed with the following command line:

```
probcli -p BOOL_AS_PREDICATE TRUE -p CLPFD TRUE -p MAXINT 127
        -p MININT -128 -p TIME_OUT T -eval_file goalFile
```

where `T` is one of the two timeouts discussed above and `goalFile` is one of the files containing a test specification.

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.

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 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 three parts. The first one shows the figures for ProB, and the last two those for {*log*} before and after the modifications described in sections 4 and 5. 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, after the extensions, {*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*}.

N	ProB				{*log*} (before extensions)				{*log*} (after extensions)			
	1 s		1 m		1 s		1 m		1 s		1 m	
	Sol	Uns	Sol	Uns	Sol	Uns	Sol	Uns	Sol	Uns	Sol	Uns
1	7	1	7	1	8		8		8		8	
2	1	1	1	1	1	1	1	1	2		2	
3	8		8			8		8	8		8	
4	17	12	17	12	7	22	7	22	17	12	17	12
5		12	10	2		12		12	10	2	10	2
6	2	2	2	2	2	2	3	1	2	2	4	
7	4		4		4		4		4		4	
8	1		1		1		1		1		1	
Totals	40	28	50	18	23	45	24	44	52	16	54	14
Time	1 m 0 s		19 m 40 s		0 m 59 s		36 m 10 s		0 m 29 s		13 m 43 s	

Table 2. Summary of the empirical results.

7 Discussion

According to [22], 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 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. Considering the results shown in [15], would also indicate that set processing would require a theory such as the one underlying $\{log\}$, and not those underlying SMT solvers. The results shown in [23,24] might conflict with the previous analysis.

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 lists to be of different lengths, but there is no constraint over their elements, $\{log\}$ may iterate over lists of, say, length one trying with different elements, but not different lengths. 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. After these modifications a new round of an empirical assessment, that was previously ran, shows that $\{log\}$ performs considerably better than three mainstream SMT solvers and quite better than ProB, in finding more test cases in less time. After these experiments we can say that $\{log\}$ should be the test case generator for Fastest and, probably, for other model-based testing tools for notations such as Z or B.

In the near future we plan to write the translator between Z to $\{log\}$ 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 TOSE **22**(11) (November 1996) 777–793
2. 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
3. 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
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. Volume 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. Softw. Test., Verif. Reliab. Online Version of Record published before inclusion in an issue – http://onlinelibrary.wiley.com/doi/10.1002/stvr.1477/abstract.
8. Cristiá, M., Frydman, C.S.: Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions. [25] 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. Volume 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.: Principles of constraint programming. Cambridge University Press (2003)
14. 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
15. Cristiá, M., Frydman, C.: Evaluation of SMT and Constraint Solvers as test case generators for the Test Template Framework. Submitted to Softw. Test., Verif. Reliab., July 2012. Available at http://www.fceia.unr.edu.ar/~mcristia/smt-cs-ttf.pdf
16. Dovier, A., Omodeo, E. G., Pontelli, E., Rossi, G.: {log}: A Language for Programming in Logic with Finite Sets. J. of Logic Programming **28**(1) (1996) 1–44.
17. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM TOPLAS **22**(5) (2000) 861–931
18. Rossi, G.: The {log} Home page. `www.math.unipr.it/~gianfr/setlog.Home.html`.
19. Dovier, A., Pontelli E., and Rossi, G.: Intensional Sets in CLP. In Palamidessi, C., ed., Logic Programming, 19th International Conference, ICLP2003, Volume 2916 of LNCS, Springer-Verlag (2003), 284-299
20. Dal Palù, A., Dovier, A., Pontelli E., and Rossi, G.: Integrating Finite Domain Constraints and CLP with Sets. In Miller, D., ed., Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, ACM Press (2003) 219–229
21. Saaltink, M.: The Z/EVES System. In Bowen, J., Hinchey, M., Till, D., eds.: ZUM '97: The Z Formal Specification Notation. (1997) 72–85

22. Leuschel, M., Butler, M.: ProB: A model checker for B. In Keijiro, A., Gnesi, S., Mandrioli, D., eds.: FME. Volume 2805 of LNCS, Springer-Verlag (2003) 855–874
23. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: SMT solvers for Rodin. [25] 194–207
24. Mentré, D., Marché, C., Filliâtre, J.C., Asuka, M.: Discharging proof obligations from Atelier B using multiple automated provers. [25] 238–251
25. 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 International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings. Volume 7316 of LNCS, Springer (2012)

Stampato in proprio, a cura degli autori, presso il Dipartimento di Matematica e Informatica dell'Università degli Studi di Parma, Parco Area delle Scienze, 53/A, 43124 Parma, adempiuti gli obblighi ai sensi della Legge n. 160 del 15.04.2004 "Norme relative al deposito legale dei documenti di interesse culturale destinate all'uso pubblico" (G.U. n. 98 del 27 aprile 2004) e del Regolamento di attuazione emanato con D.P.R. n. 252 del 3 maggio 2006 (G.U. n. 191 del 18 agosto 2006) entrato in vigore il 2 settembre 2006 [precedente normativa abrogata: Legge n. 374 del 2.2.1939, modificata in D.L. n. 660 del 31 agosto 1945].

Esemplare fuori commercio per il deposito legale agli effetti della legge 15 aprile 2004, n. 160.