$\{log\}$ Applications to Software Specification and Verification

Maximiliano Cristiá and Gianfranco Rossi

Contents

1 $\{log\}$ installation

$\{log\}$ ('setlog') is a constraint logic programming language. Besides it's a *satisfiability solver* and as such it can be used as an automated theorem prover. One of $\{log\}$'s distinctive features is that sets are first-class entities of the language.

$\{log\}$ was first developed by Gianfranco Rossi and his PhD students in Italy during the mid '90. Since 2012 Gianfranco Rossi and Maximiliano Cristiá work together in extending $\{log\}$ to binary relations and related theories.

$\{log\}$ is a Prolog program. Then, you first need to install a Prolog interpreter. So far $\{log\}$ runs only on SWI-Prolog (`http://www.swi-prolog.org`). After installing SWI-Prolog you must download $\{log\}$ and all the library files from here:

`http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html`

This document is focused on showing how Z specifications can be translated into $\{log\}$ and, later, on how $\{log\}$ can be used to run simulations and automated proofs. This can help users of other specification languages such as B and VDM to use $\{log\}$ along the same lines. The presentation is rather informal and user-oriented. More technical and formal presentations can be found in the papers published by the authors. We also assume the reader has at least a basic knowledge of the Z notation.

2 Translating Z specifications into $\{log\}$

Many Z specifications can be easily translated into $\{log\}$. This means that $\{log\}$ can serve as a programming language in which a prototype of a Z specification can be immediately implemented.

We will consider Z specifications of a very specific form. Here, we will show how these Z specifications can be translated into $\{log\}$. To that end we will use a running example. Later on we will explain with some detail how Z elements not appearing in the example can be translated into $\{log\}$; we will see that some Z elements can be translated in more than one way.

2.1 The running example

The specification to be used as running example is one of the classic Z specifications first used by Spivey in several articles and books [1]. It's known as the *birthday book*. It's a system which records people's birthdays, and is able to issue a reminder when the day comes round.

2.1.1 The Z specification

In our account of the system, we need to deal with people's names and with dates. Then, we introduce the following basic types.

$[NAME, DATE]$

Now we can define the state schema of the specification as follows.

```
┌─ BirthdayBook ──────────────────
│ known : ℙ NAME
│ birthday : NAME ⇸ DATE
└─────────────────────────────────
```

where *known* is the set of names with birthdays recorded; and *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.

The initial state of the birthday book is the following.

$$\begin{array}{|l}
\hline \textit{BirthdayBookInit} \\
\textit{BirthdayBook} \\
\hline
\textit{known} = \emptyset \\
\textit{birthday} = \emptyset \\
\hline
\end{array}$$

The following schema describes the predicates that should be state invariants.

$$\begin{array}{|l}
\hline \textit{BirthdayBookInv} \\
\textit{BirthdayBook} \\
\hline
\textit{known} = \operatorname{dom} \textit{birthday} \\
\hline
\end{array}$$

As can be seen, the value of *known* can be derived from the value of *birthday*. This makes *known* a *derived* component. It would be possible to specify the system without mentioning *known* at all. However, giving names to important concepts helps to make specifications more readable. The specification doesn't commit the programmer to represent *known* explicitly in an implementation.

The first operation we specify is how to add a birthday to the birthday book. As always we first model the normal behavior, then the abnormal behaviors and finally we assemble all the schemas in a single schema expression.

$$\begin{array}{|l}
\hline \textit{AddBirthdayOk} \\
\Delta \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\hline
\textit{name}? \notin \textit{known} \\
\textit{known}' = \textit{known} \cup \{\textit{name}?\} \\
\textit{birthday}' = \textit{birthday} \cup \{\textit{name}? \mapsto \textit{date}?\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{NameAlreadyExists} \\
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\hline
\textit{name}? \in \textit{known} \\
\hline
\end{array}$$

$$\textit{AddBirthday} == \textit{AddBirthdayOk} \lor \textit{NameAlreadyExists}$$

The second operation to be specified is the one that shows the birthday of a given person.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{FindBirthdayOk}} \\
\hline
\Xi \mathit{BirthdayBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{date}! : \mathit{DATE} \\
\hline
\mathit{name}? \in \mathit{known} \\
\mathit{date}! = \mathit{birthday}(\mathit{name}?) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NotAFriend}} \\
\hline
\Xi \mathit{BirthdayBook} \\
\mathit{name}? : \mathit{NAME} \\
\hline
\mathit{name}? \notin \mathit{known} \\
\hline
\end{array}$$

$$\mathit{FindBirthday} == \mathit{FindBirthdayOk} \lor \mathit{NotAFriend}$$

Finally we have an operation listing all the persons whose birthday is a given date.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Remind}} \\
\hline
\Xi \mathit{BirthdayBook} \\
\mathit{today}? : \mathit{DATE} \\
\mathit{cards}! : \mathbb{P}\,\mathit{NAME} \\
\hline
\mathit{cards}! = \mathrm{dom}(\mathit{birthday} \rhd \{\mathit{today}?\}) \\
\hline
\end{array}$$

2.1.2 The {*log*} code

The {*log*} code of the translation of the Z specification should be saved in a file with extension `.slog`. It is convenient to put this file in the same folder where {*log*} was installed.

Most Z schemas are translated into {*log*} *clauses*. A {*log*} clause is a sort of subroutine or subprogram or procedure of a regular programming language. Each clause can receive zero or more arguments. When a Z schema representing an operation is translated, the corresponding clause receives as arguments all the state variables, all the input variables and all the output variables.

In {*log*} variables must always begin with an uppercase letter or the underscore character (`_`), although this is usually saved for special cases. Any identifier beginning with a lowercase letter is a constant. The prime character ($'$) cannot be used as part of a variable's name. Therefore, in order to denote the after state variables we will put the underscore character at the end ('`_`'). Then, for instance, the state variables of the birthday book will be `Known` and `Birthday`; and those of the after state will be `Known_` and `Birthday_`. In the same way, ? y ! cannot be part of variables' names in {*log*}. In these cases we will use `_i` and `_o` as suffixes denoting input and output variables, respectively. Then, for example, *name*? (see schema *AddBirthdayOk*) becomes `Name_i` in {*log*}; and *cards*! (see schema *Remind*) becomes `Cards_o`.

Hence, the interface of the {*log*} clause corresponding to the Z operation named *AddBirthday* is the following:

```
addBirthday(Known,Birthday,Name_i,Date_i,Known_,Birthday_)
```

where `Name_i` and `Date_i` correspond to input variables *name*? and *date*? declared in *AddBirthday*. Note that althugh *AddBirthday* begins with an uppercase letter `addBirthday` begins with a lowercase letter because $\{log\}$ clauses must begin in that way; on the other hand, although *name*? and *date*? begin with lowercase letters, `Name_i` and `Date_i` begin with uppercase letters because we want them to variable arguments. Finally, `Known` and `Birthday` represent the before state while `Known_` y `Birthday_` represent the after state.

Now we can give the definition of the `addBirthday` clause:

```
addBirthday(Known,Birthday,Name_i,Date_i,Known_,Birthday_) :-
  addBirthdayOk(Known,Birthday,Name_i,Date_i,Known_,Birthday_)
  or
  nameAlreadyExists(BirthdayBook,Name_i,BirthdayBook_).
```

where `or` denotes logical disjunction. Observe that `Date_i` isn't passed to `nameAlreadyExists` as an argument because it isn't declared in *NameAlreadyExists*; besides, look that the clause ends in a dot.

In turn the definition of `addBirthdayOk` is the following:

```
addBirthdayOk(Known,Birthday,Name_i,Date_i,Known_,Birthday_) :-
  Name_i nin Known &
  un(Known,{Name_i},Known_) &
  un(Birthday,{[Name_i,Date_i]},Birthday_).
```

where:

- The `&` symbol denotes logical conjunction.
- `nin` corresponds to the $\notin$ operator. That is, `Name_i nin Known` means $Name\_i \notin Known$.
- `un(Known,{Name_i},Known_)` means $Known\_ = Known \cup \{Name\_i\}$.
- The last statement implements the state change.

Let's now see the definition of `nameAlreadyExists`:

```
nameAlreadyExists(Known,Birthday,Name_i,Known_,Birthday_) :-
  Name_i in Known &
  Known_ = Known &
  Birthday_ = Birthday.
```

where we can see how to say that there's no state change.

The order in which clauses are given is irrelevant in the sense that a clause can be defined before or after of being invoked for the first time.

The next schema to be translated is *FindBirthday*.

```
findBirthday(Known,Birthday,Name_i,Date_o,Known_,Birthday_) :-
  findBirthdayOk(Known,Birthday,Name_i,Date_o,Known_,Birthday_)
  or
  notAFriend(Known,Birthday,Name_i,Known_,Birthday_).
```

where we can see that the translation starts to be repetitive. Now we show the definition of `findBirthdayOk` and `notAFriend`:

```
findBirthdayOk(Known,Birthday,Name_i,Date_o,Known_,Birthday_) :-
  Name_i in Known &
  apply(Birthday,Name_i,Date_o) &
  Known_ = Known &
  Birthday_ = Birthday.

notAFriend(Known,Birthday,Name_i,Known_,Birthday_) :-
  Name_i nin Known &
  Known_ = Known &
  Birthday_ = Birthday.
```

where we can see how to use output variables. We can also see the $\{log\}$ `apply` predicate which implements function application. That is, `apply(F,X,Y)` is true if and only if $F(X) = Y$ holds. Note that `apply(F,X,Y)` makes sense only if `F` is a function and not any set. $\{log\}$ enforces this because, internally, `apply` "types" `F` with the predicate `pfun(F)`. Hence, if `apply` is called with a first argument that happens not to be a function, the predicate will fail making $\{log\}$ to answer `no`.

Finally the translation of *Remind* is the following:

```
remind(Known,Birthday,Today_i,Cards_o,Known_,Birthday_) :-
  rres(Birthday,{Today_i},M) &
  dom(M,Cards_o) &
  Known_ = Known &
  Birthday_ = Birthday.
```

This is an interesting example because it shows how set and relational expressions must be translated. Given that in $\{log\}$ set and relational operators are implemented as predicates, it's impossible to write set and relational expressions. Instead, we have to introduce new variables (such as `M`) to "chain" the predicates. Predicate `rres(R,A,S)` stands for $S = R \rhd A$ while `dom(R,B)` stands for $\mathrm{dom} R = B$ (that is, `dom(R,B)` allows to compute the domain of `R` and "save" it in `B`).

The schema specifying the initial state is translated as a clause receiving the state variables:

```
birthdayBookInit(Known,Birthday) :-
  Known = {} &
  Birthday = {}.
```

where `{}` denotes the empty set ($\emptyset$).

The translation of *BirthdayBookInv* is given in Section 4.

2.2 Types in $\{log\}$

So far we haven't given the types of the variables. $\{log\}$ is essentially an untyped formalism but in the last version we have added a type system similar to Z's. $\{log\}$'s type system is described in detail in chapter 9 of $\{log\}$ user's manual. Here we will give a broad description of how to use types in $\{log\}$.

$\{log\}$'s type system allows users to define type synonyms to simplify the type declaration of clauses and variables. For example, we can define the following type synonyms:

```
:- dec_type(bb,stype([name,date])).
:- dec_type(kn,stype(name)).
```

where `bb` is a type identifier o synonym of the type `stype([name,date])`. In `stype([name,date])`, `name` and `date` correspond to the basic types *NAME* and *DATE* of the Z specification. In $\{log\}$, Z basic types (called *uninterpreted types* in $\{log\}$) can be introduced without any previous declaration. In $\{log\}$ basic types must begin with a lowercase letter (i.e. they are constants). Besides, `[name,date]` corresponds to the Cartesian product between `name` and `date` (i.e., is equivalent to $\mathit{NAME} \times \mathit{DATE}$ in

Z). In turn, `stype([name,date])` corresponds to the type of all the sets of type `[name,date]` (i.e., is equivalent to $\mathbb{P}(\mathit{NAME} \times \mathit{DATE})$ in Z). Hence, `stype([name,date])` corresponds to the type of all the binary relations between `name` and `date` (i.e., $\mathit{NAME} \leftrightarrow \mathit{DATE}$ in Z).

These type synonyms allow us to declare the type of the `addBirthdayOk` clause:

```
:- dec_p_type(addBirthdayOk(kn,bb,name,date,kn,bb)).
```

This declaration must come before the clause definition:

```
:- dec_p_type(addBirthdayOk(kn,bb,name,date,kn,bb)).
addBirthdayOk(Known,Birthday,Name_i,Date_i,Known_,Birthday_) :-
  Name_i nin Known &
  ...
```

The `dec_p_type` predicate has only one argument of the following form:

```
clause_name(parameters)
```

In turn, `parameters` is a list whose elements corresponds one-to-one to the clause arguments. In this way, the type of each clause argument is given. Then, the type of `Known` is `kn`, the type of `Birthday` is `bb`, etc.

In this version of $\{log\}$ clauses must be defined before they are used, due to the introduction of types. We remark this because this wasn't the case in previous versions of $\{log\}$ and isn't the case in Prolog programs. This means that we should first define `addBirthdayOk` or `nameAlreadyExists` and then `addBirthday` because the latter invokes the former two.

Note that above we didn't respect this rule because we decided to introduce types later for pedagogy reasons. From now on we will proceed according to the above rule; that is, by defining the clauses before they are used

The following is the typed version of the `remid` clause.

```
:- dec_p_type(remind(kn,bb,date,kn,kn,bb)).
remind(Known,Birthday,Today_i,Cards_o,Known_,Birthday_) :-
  rres(Birthday,{Today_i},M) & dec(M,bb) &
  dom(M,Cards_o) &
  Known_ = Known &
  Birthday_ = Birthday.
```

This clause is interesting because it shows how clause variables are typed by means of the `dec(V,t)` predicate. Indeed, `dec(V,t)` is interpreted as "variable `V` is of tye `t`".

The $\{log\}$ code including type declarations of the complete translation of the birthday book can be found in Appendix A.

Recall that partial functions *aren't* a type in Z. The same happens in $\{log\}$; in fact it is impossible to define the type of all partial functions. The natural numbers are another example of a set that isn't a type. This means that if in Z we have $f : X \pfun Y$ in $\{log\}$ we declare `F` to be of type `stype([x,y])` and then we should either establish or prove that `F` is a function (we'll see further details on this later). Likewise, if in Z we declare $x : \mathbb{N}$ in $\{log\}$ we must declare `X` to be of type `int` and then either establish or prove that `0 =< X` holds. In general, when a Z specification is translated into $\{log\}$ it would be convenient to first normalize the Z specification and then start the translation into $\{log\}$. In this case the Z types are translated straightforwardly and the predicates introduced due to the normalization process become constraints at the $\{log\}$ level (i.e. `0 =< X`) or they are proved to be invariants. For instance, $x : \mathbb{N}$ is a non-normalized declaration because $\mathbb{N}$ isn't a type (it's a set). The normalized declaration would be $x : \mathbb{Z}$ plus $x \geq 0$ conjoined in the predicate part. In this case, in $\{log\}$ the type of x is `int` and we should either establish or prove that x is always greater than or equal to zero.

In the birthday book the declaration $birthday : NAME \pfun DATE$ isn't normalized. The normalized declaration would be $birthday : NAME \leftrightarrow DATE$ plus $birthday \in NAME \pfun DATE$ in the predicate part. In this case, *birthday*'s type is translated as we did above and we could add `pfun(Birthday)` to the clauses where we use `Birthday`. `pfun` is a $\{log\}$ predicate implementing the definition of (partial) function as a subclass of the sets of ordered pairs. But then, why we haven't state `pfun(B)` somewhere in `addBirthday`? Because the idea is to show how $\{log\}$ can be used to prove that `pfun(B)` is a state invariant of the specification. In fact had we written the state schema *BirthdayBook* with normalized declarations, it would have resulted as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBook}} \\ \hline
known : \mathbb{P}\,NAME \\
birthday : NAME \leftrightarrow DATE \\ \hline
birthday \in NAME \pfun DATE \\ \hline
\end{array}$$

where $birthday \in NAME \pfun DATE$ would be a state invariant by definition (recall Section 6 of "Introduction to the Z notation"). Therefore, we should have written the schema as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBook}} \\ \hline
known : \mathbb{P}\,NAME \\
birthday : NAME \leftrightarrow DATE \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBookInv}} \\ \hline
BirthdayBook \\ \hline
known = \operatorname{dom} birthday \\
birthday \in NAME \pfun DATE \\ \hline
\end{array}$$

where it's clear that we must *prove* that `pfun(B)` is a state invariant of the specification. We'll come back to this point in Section 4.

2.3 An alternative to encoding state variables

The state schema of the birthday book specification has two variables (*known* and *birthday*). These variables are included as parameters in the clauses encoding the operations of the specification; besides, the after state variables must also be included (those ending with '`_`'). This can become complex if the Z specification declares many state variables. In this case one should define $\{log\}$ clauses with many parameters which would make the code hard to read.

State variables can also be encoded by bundling them in a Prolog list. Later Prolog unification is used to access each state variable. The following $\{log\}$ code uses this encoding in `addBirthdayOk`[1]:

[1]Here we don't include type declarations to simplify the presentation.

```
addBirthdayOk(BirthdayBook,Name_i,Date_i,BirthdayBook_) :-
  BirthdayBook = [Known,Birthday] &
  Name_i nin Known &
  un(Known,{Name_i},Known_) &
  un(Birthday,{[Name_i,Date_i]},Birthday_) &
  BirthdayBook_ = [Known_,Birthday_].
```

where `BirthdayBook` represents the before state and `BirthdayBook_` the after state. Both variables are assumed to be lists of length two. This assumption is enforced by means of the equalities:

```
BirthdayBook = [Known,Birthday]
BirthdayBook_ = [Known_,Birthday_]
```

This means that if for any reason `addBirthdayOk` is invoked with a first parameter that doesn't unify with `[Known,Birthday]`, the invocation will fail (same with the second parameter).

Observe how the unification between `BirthdayBook_` with the list `[Known_, Birthday_]` sets the after state. See that the list contains the same variables used in the postcondition of the operation.

It is possible to unify `BirthdayBook` with a list containing any two variables; they need not to be the same used in the specification. For example, `BirthdayBook = [K,B]`.

Note that the first and last arguments of `addBirthdayOk` are named as the state schema of the Z specification (i.e. *BirthdayBook*). This convention helps to relate the Z specification with the $\{log\}$ code, but it is in no way mandatory.

Clearly, encoding state variables in this way carries the complexity of a large number of arguments to the length of the list representing the state of the system. Hence, it is at the programmer discretion what is the best encoding for each situation. In fact, both encodings can coexist. In this case care must be taken when invoking clauses.

A slight advantage of using the encoding with Prolog lists can be seen in operations not using some of the state variables. For instance, the Z operation named *Remind* doesn't use state variable *known*. In that case it can be encoded as follows:

```
remind(BirthdayBook,Today_i,Cards_o,BirthdayBook_) :-
  BirthdayBook = [_,B] &
  rres(B,{Today_i},M) &
  dom(M,Cards_o) &
  BirthdayBook_ = BirthdayBook.
```

where we have written '`_`' in place of the first element of the list unifying with `BirthdayBook`. This means that this element of the list doesn't matter; it can be anything. If the list of state variables has many elements and some of them are not used in a clause, the programmer can use '`_`' in different positions instead of writing variable names.

Another benefit of using the list-based encoding is when stating that no state change occurs. For example, in the birthday book specification `BirthdayBook_ = BirthdayBook` is written instead of `Known_ = Known & Birthday_ = Birthday`. This encoding is a good help when there are many state variables.

If the list of state variables is long it might be a problem recalling what is the position of each variable. Switching the order of two or more variables will lead to unsound programs.

2.4 Translating ordered pairs

Ordered pairs are encoded as Prolog lists of two elements. For instance, if x is a variable $(x, 3)$ is translated as `[X,3]`.

The translation of $p : \mathbb{Z} \times V$ and $p.1 = x - 4$ is: `P = [A,_] & A is X - 4` (see the `is` operator in Section 2.7). This means that, basically, we use unification to force that `P` be a two elements list such that its first element is variable `A`, which in turn must be equal to `X - 4`. `A` must be a variable name not used in the clause. Note that we write '`_`' as the second argument because we aren't interested in the second component of `P`.

Prolog lists must not be used to encode Z sequences.

2.5 Translating sets

2.5.1 Extensional sets — Introduction to set unification

The set $\{1,2,3\}$ is simply translated as `{1,2,3}`. If one of the elements of the set is a variable or an element of an enumerated type, take care of the differences concerning variables and constants in Z and $\{log\}$. For example, if in Z x is a variable, then the set $\{2,x,6\}$ is translated as `{2,X,6}`.

However, $\{log\}$ provides a form of extensional sets that, in a sense, is more powerful than the one offered by Z. The term `{.../...}` is called *extensional set constructor*. In `{E/C}` the second argument (i.e. `C`) must be a set. `{E/C}` means $\{E\} \cup C$. Then, there are solutions where $E \in C$. To avoid such solutions (in case they're incorrect or unwanted) the predicate $E \notin C$ must be explicitly added to the formula. In order to make the language more simple, $\{log\}$ accepts and prints terms such as `{1,2 / X}` instead of `{1 / {2 / X}}`.

The extensional set constructor is useful and in general it's more efficient than other encodings. For example, the Z predicate:

$$A' = A \setminus \{d?\}$$

can be translated by means of the $\{log\}$ predicate `diff`, whose semantics is equivalent to $\setminus$ (see Table 1):

```
diff(A,{D_i},A_)
```

Bu it also can be translated by means of an extensional set:

```
A = {D_i / A_} & D_i nin A_ or D_i nin A & A_ = A
```

which in general is more efficient.

That is, the predicate `A = {D_i / A_}` *unifies* `A` with `{D_i / A_}` in such a way that it finds values for the variable to make the equality true. If such values don't exist the unification fails and $\{log\}$ tries the second disjunct.

Why we conjoined `D_i nin A_`? Simply because, for instance, `A = {1,2}`, `D_i = 1` and `A_ = {1,2}` is a solution of the equation but it isn't a solution of $A' = A \setminus \{d?\}$. Precisely, when `D_i nin A_` is conjoined all the solutions where `D_i` belongs to `A_` are eliminated.

$\{log\}$ solves equalities of the form `B = C`, where `B` and `C` are terms denoting sets, by using *set unification*. Se unification is at the base of the deductive power of $\{log\}$ making it an important extension of Prolog's unification algorithm. Set unification is inherently computationally hard because finding out whether or not two sets are equal implies, in the worst case, computing all the permutations of their elements. On top of that, it is the fact that $\{log\}$ can deal with *partially specified* sets, that is sets where some of their elements or part of the set are variables. For these reasons, in general, $\{log\}$ will show efficiency problems when dealing with certain formulas but, at the same time, we aren't aware of other tools capable to solve some of the problems $\{log\}$ can.

2.5.2 Cartesian products

In $\{log\}$ Cartesian products are written `cp(A,B)` where `A` and `B` can be variables, extensional sets and Cartesian products.

2.5.3 Integer intervals

A Z integer interval such as $n \mathinner{.\,.} m$ is translated as `int(n,m)`.

2.6 Translating function application

One interesting application of set unification is the application of a function to its argument. Given that partial functions are frequently used in Z it's necessary to add predicates of the form $x \in \operatorname{dom} f$, before attempting to apply f to x. The translation of these formulas into $\{log\}$ can be done by using the predicate `apply` or by using a set membership predicate which leads to set unification. For example the Z formula:

$$x \in \operatorname{dom} f \wedge f(x) = y$$

can be translated in a direct fashion:

```
dom(F,D) & X in D & apply(F,X,Y)
```

or using a set membership predicate:

```
[X,Y] in F & pfun(F)
```

which is equivalent and more efficient as it doesn't require to "compute" the domain of `F`. Concerning the equivalence of both formulas, note that `[X,Y] in F` is transformed into `F = {[X,Y] / G}` (for some `G`). Then, if there are no ordered pairs in `F` whose first component is `X`, the unification will fail which is equivalent to say that $x \in \operatorname{dom} f$ is false. In the same way, if there's an ordered pair in `F` whose first component is `X` but whose second component is not `Y`, the unification will fail as well (which is equivalent to $f(x) \neq y$). Variable `G` is automatically generated by $\{log\}$ in such a way that it will be new in the formula. This is interpreted as "there exists `G` such that…".

However, any of the encodings using `pfun` might not be correct. In effect, the fact that `F` is a function might be something that should be proved to hold, usually in the form of a state invariant (recall schema *BirthdayBookInv* shown above and the discussion around it). Consequently, if we use `apply` or `pfun` we would be establishing that `F` is a function rather than getting it as a consequence of the encoding. In other words, if we write `pfun(F)` we would be indicating to the programmer to control that the data structure used to encode `F` be a function every time that function is applied to an argument. Usually, the most reasonable implementation is for the code to guarantee that the data structure is a function without explicitly controlling that.

Therefore, we can't include `apply(F,...)` nor `pfun(F)` in the operations of the model if `pfun(F)` is an invariant to be proved because otherwise we would be cheeting. Then, the question is, how $f\,x = y$ is translated when we only know that f is a binary relation and that $x \in \operatorname{dom} f$? The $\{log\}$ code is the following:

```
F = {[X,Y] / G} & [X,Y] nin G & comp({[X,X]},G,{})
```

The justification is as follows. If we know that $x \in \operatorname{dom} f$ the there exist `Y` and `G` such that `F = {[X,Y] / G} & [X,Y] nin G`, due to the above analysis. Besides, if we are saying that we can apply f to x is because there is one and only one ordered pair in f whose first component is x. Note that we aren't saying that f is a function, we're just saying that f is *locally* a function in x (it might well be a function in other points of its domain but we don't know that yet). Saying that in f there is exactly one ordered pair whose first component is x is the same than saying that there are no ordered pairs in `G` whose first component is x. We say this by using the composition operator defined over binary relations, namely `comp` (see Table 2), when we conjoin the predicate `comp({[X,X]},G,{})`. Indeed, this predicate says that when `{[X,X]}` is composed with `G` the result is the empty set. This can happen for two reasons: `G` is the empty binary relation, in which case it's obvious that there are no ordered pairs with first component `X`; or `G` is non-empty but no pair in it composes with `[X,X]`, which is equivalent to say that `X` does not belong to the domain of `G`. We could have said the same by stating that `dom(G,D) & X nin D` but this is usually less efficient because it requires to compute the domain of `G`.

The $\{log\}$ library `setloglibpf.slog` defines the predicate `applyTo(F,X,Y)` which implements the code shown above. This library can be loaded with `add_lib('setloglibpf.slog')`.

Observe that in `findBirthdayOk` we have used `apply` which, after the above analysis, is incorrect because `pfun(Birthday)` is intended to be an invariant of the program. We should replace `apply` by `applyTo`. We didn't do it in that way because we think that it requires a rather complex explanation when we were just introducing $\{log\}$.

2.7 Translating arithmetic expressions

Almost all Z arithmetic expressions are translated directly into $\{log\}$, with some exceptions. The relational symbols $\leq$, $\geq$ and $\neq$ are translated as `=<`, `>=` and `neq`, respectively. The arithmetic operators are the usual ones: `+`, `-`, `*`, `div` y `mod`.

An equality of the form $x' = x + 1$ is translated as `X_ is X + 1` (that is, in arithmetic equalities you mustn't use `=` but `is`). Furthermore, if in Z we have $A = \{x, y - 4\}$ (A, x and y variables) it has to be encoded as: `A = {X,Z} & Z is Y - 4`, where `Z` is a variable not used in the clause. The problem is that nor $\{log\}$ nor Prolog evaluate arithmetic expression unless the programmer forces it by using the `is` operator. This means that if in $\{log\}$ we run `{X,Y - 4} = {Y - 3 - 1,X}`, the answer will be `no` because $\{log\}$ will try to find out whether or not `Y - 4 = Y - 3 - 1` without evaluating the expressions (that is, it will consider them, basically, as character strings where `Y` is an integer variable and thus it is impossible for the equality to hold regardless of the value of `Y`). On the contrary, if we run `{X,A} = {B,X} & A is Y - 4 & B is Y - 3 - 1` $\{log\}$ will return several solutions (with some repetitions), meaning that the sets are equal in several ways.

The same applies to the `neq` predicate: for $\{log\}$ `Y - 4 neq Y - 3 - 1` is true. As a consequence we must write: `H is Y - 4 & U is Y - 3 - 1 & H neq U`. However, this is not necessary with the order predicates: `X + 1 > X` is satisfiable but `X - 1 > X` isn't.

On the other hand, $A \subseteq \mathbb{N}$ or $A : \mathbb{P}\mathbb{N}$ are translated with a restricted universal quantifier (see Section 2.9.1):

```
foreach(X in A, 0 =< X)
```

In any case, usually, $A \subseteq \mathbb{N}$ and $A : \mathbb{P}\mathbb{N}$ are invariants. If this is the intention, then they should be proved to be rather than establishing them.

2.8 Translating set operators

Set, relational, functional and sequence operators are translated as shown in Tables 1, 2 and 3.

In order to be able to work with the sequence operators shown in Table 3 load the corresponding library file (e.g. `consult('setlogliblist.slog')`) into the $\{log\}$ environment.

OPERATOR	$\{log\}$	MEANING
set	`set(A)`	A is a set
equality	`A = B`	$A = B$
set membership	`x in A`	$x \in A$
union	`un(A,B,C)`	$C = A \cup B$
intersection	`inters(A,B,C)`	$C = A \cap B$
difference	`diff(A,B,C)`	$C = A \setminus B$
subset	`subset(A,B)`	$A \subseteq B$
strict subset	`ssubset(A,B)`	$A \subset B$
disjointness	`disj(A,B)`	$A \parallel B$
cardinality	`size(A,n)`	$\lvert A\rvert = n$
NEGATIONS		
equality	`A neq B`	$A \neq B$
set membership	`x nin A`	$x \notin A$
union	`nun(A,B,C)`	$C \neq A \cup B$
intersection	`ninters(A,B,C)`	$C \neq A \cap B$
difference	`ndiff(A,B,C)`	$C \neq A \setminus B$
subset	`nsubset(A,B)`	$A \not\subseteq B$
disjointness	`ndisj(A,B)`	$A \not\parallel B$

Table 1: Set operators available in $\{log\}$

OPERATOR	$\{log\}$	MEANING
binary relation	`rel(R)`	R is a binary relation
partial function	`pfun(R)`	R is a partial function
function application	`apply(f,x,y)`	$f(x) = y$
domain	`dom(R,A)`	$\operatorname{dom} R = A$
range	`ran(R,A)`	$\operatorname{ran} R = A$
composition	`comp(R,S,T)`	$T = R \circ S$
inverse	`inv(R,S)`	$S = R^{-1}$
domain restriction	`dres(A,R,S)`	$S = A \lhd R$
domain anti-restriction	`dares(A,R,S)`	$S = A \ntriangleleft R$
range restriction	`rres(A,R,S)`	$S = R \rhd A$
range anti-restriction	`rares(A,R,S)`	$S = R \ntriangleright A$
update	`oplus(R,S,T)`	$T = R \oplus S$
relational image	`rimg(R,A,B)`	$B = R[A]$
NEGACIONES		
All negations are written by prefixing a letter `n` to the corresponding operator. For example, the negation of `dom(R,A)` is `ndom(R,A)`, that of `dares(A,R,S)` is `ndares(A,R,S)`, etc.		

Table 2: Relational operators available in $\{log\}$

OPERATOR	$\{log\}$	MEANING
sequence	`slist(s)`	s is a sequence
extensional sequence	`{[1,a],[2,b],...,[n,z]}`	$\langle a,b,\dots,z\rangle$
head	`head(s,e)`	$e = head\, s$
tail	`tail(s,t)`	$t = tail\, s$
last	`last(s,e)`	$e = last\, s$
front	`front(s,t)`	$t = front\, s$
add (cons)	`add(s,e,t)`	$t = s \frown \langle e\rangle$
concatenation	`concat(s,t,u)`	$u = s \frown t$
filter	`filter(A,s,t)`	$t = A \upharpoonright s$
extraction	`extract(s,A,t)`	$t = s \restriction A$

Table 3: Sequence operators available in $\{log\}$

The cardinality operator accepts as second argument only a constant or a variable. Hence, if we run `size(A,X + 1)` $\{log\}$ answers `no`; instead if we run `size(A,Y) & Y is X + 1` (`Y` must be a variable not used in the clause) the answer is `true` because the formula is satisfiable. $\{log\}$ will answer `no` if we execute `size(A,Y) & Y = X + 1`.

2.9 Translating logical operators

Only logical conjunction (`&`) and disjunction (`or`) are available in $\{log\}$. Logical negation ($\neg$) doesn't exist because it's replaced by the negations of the operators available in $\{log\}$. For instance, if we want to translate $\neg\, x \in A$ we write in $\{log\}$ `X nin A`. In the same way, $\neg\, A = b$ is translated as `A neq b`. In general, for each set, relational and arithmetic operator there exists a $\{log\}$ predicate implementing its negation. For instance, the Z predicate $A \not\subseteq B$ is translated as `nsubset(A,B)`; and $\neg\, a \leq y$ as `A > Y`. Tables 1 and 2 include the negation for every set theoretic operator.

As a summary, a Z schema like the following one:

WithdrawE
$\Xi Bank$
$n? : NIC$
$a? : MONEY$
$msg! : MSG$

$\neg\,(n? \in \operatorname{dom} sa \land a? \leq sa(n?) \land a? > 0)$
$msg! = error$

can be translated as follows (we assume that schema *Bank* only declares variable *sa*):

```
withdrawE(Sa,N_i,A_i,Msg_o,Sa_) :-
  (dom(Sa,D) &
   N_i nin D
   or
   apply(Sa,N_i,Y) &
   A_i > Y
   or
   A_i =< 0
  ) &
  Msg_o = error &
  Sa_ = Sa.
```

The remaining propositional connectives can be encoded by means of the well-known propositional equivalencies in terms of $\wedge$, $\vee$ and $\neg$. For example, $p \Rightarrow q$ is first written as $\neg\, p \vee q$ and then $\neg\, p$ and q are translated into $\{log\}$.

2.9.1 Quantifiers

In general existential quantifiers need not to be translated because $\{log\}$ semantics is based on existentially quantifying all variables of any given program. For example, if in Z we have:

$$\exists x : \mathbb{N} \mid x \in A$$

it can be translated as:

```
0 =< X & X in A
```

because the semantics of the $\{log\}$ program is, essentially, an existential quantifier over both variables.

Things are different when dealing with universal quantifiers. In $\{log\}$ we only have so-called *restricted universal quantifiers* (RUQ). A RUQ is a formula of the following form:

$$\forall x \in A : P(x)$$

where P is a proposition depending on x. In $\{log\}$ the simplest RUQ are encoded as follows:

```
foreach(X in A,P(X))
```

There are more complex and expressive RUQ available[2].

Recall that a proper use of the Z language avoids most of the quantified formulas.

2.10 Translating axiomatic definitions

There's no simple way of translating all axiomatic definitions into $\{log\}$ because they are so general and serve to so many purposes. For this reason we will show how to translate the most used forms of axiomatic definitions.

One of the problems when translating axiomatic definitions is that they are global objects while in $\{log\}$ global objects don't really exist.

The easiest and simplest way of translating axiomatic definitions is defining a $\{log\}$ clause including all the predicates of the axiomatic definition. Here we're assuming that each axiomatic definition declares only one variable. As always, all the restrictions on constant and variable names as well as the considerations on translating types, must be carefully watched.

Let's see the most common cases.

Z specification.

[2]Have a look at chapter 6 of $\{log\}$ user's manual and then ask for help to the instructor.

$root : USR$

$\{log\}$ code.

```
:- dec_p_type(root_da(usr)).
root_ad(X) :- X = root.
```

Z specification.

$adm : \mathbb{P}\, USR$

$adm = \{root, sec\}$

$\{log\}$ code.

```
:- dec_p_type(adm_da(stype(usr))).
adm_ad(X) :- X = {root,sec}.
```

As shown in the following example, the translation of some axiomatic definitions uses $\{log\}$ recursive definitions.

Z specification.

$sum : \mathrm{seq}\,\mathbb{Z} \rightarrow \mathbb{Z}$

$sum\,\langle\rangle = 0$

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

$\{log\}$ code. Recall that lists are sets of ordered pairs.

```
:- dec_p_type(sum_da(stype([int,int]),int))).
sum_ad({},0).
sum_ad({[N,X]/S},Sum) :-
  [N,X] nin S & Sum is X + Sum1 & sum_ad(S,Sum1).
```

Note that the type of `sum_da` is declared only once.

These clauses can be invoked from clauses implementing schemas or other axiomatic definitions. For example, the following is part of the translation of an operation specifying how user *a* creates the system account of user *u*:

```
createUsr(...,A,U,...) :- ... & root_ad(A) & ...
```

which means that we're checking that user `A` is *root*. The following version checks that `A` is an administrator:

```
createUsr(...,A,U,...) :- ... & adm_ad(Adm) & A in Adm...
```

where `Adm` is a new variable. In these cases variable `A` *unifies* with the argument named `X` in `root_ad` and `adm_ad`. In the first case this implies that `A` is equal to `root`. In the second case this implies that `A` is equal to `{root,sec}`.

3 Simulating $\{log\}$ prototypes

A $\{log\}$ implementation of a Z specification is easy to get but usually it won't meet the typical performance requirements demanded by users. Hence, we see a $\{log\}$ implementation of a Z specification more as a *prototype* than as a final program. On the other hand, given the similarities between a Z specification and the corresponding $\{log\}$ program, it's reasonable to think that the prototype is a *correct* implementation of the specification[3]. Then, we can use these prototypes to make an early validation of the requirements.

Validating user requirements by means of prototypes entails executing the prototypes together with the users so they can agree or disagree with the behavior of the prototypes. This early validation will detect many errors, ambiguities and incompleteness present in the requirements and possible misunderstandings or misinterpretations generated by the software engineers. Without this validation many of these issues would be detected in later stages of the project thus increasing the project costs. Think that if one of these issues is detected once the product has been delivered it means to correct it from the requirements document, the specification, the design, the implementation, the user documentation, etc.

Since we see $\{log\}$ programs as prototypes we talk about *simulations* or *animations* rather than *executions* when speaking about running them. However, technically, what we do is no more than running a program. The word *simulation* is usually used in the context of *models* (e.g. modeling and simulation). In a sense, our $\{log\}$ programs are *executable models* of the user requirements. On the other hand, the word *animation* is usually used in the context of formal specifications. In this sense, the $\{log\}$ implementation of a Z specification can be seen as an *executable specification*. In fact, as we will see, $\{log\}$ programs have features and properties usually enjoyed by specifications and models, which are rare or nonexistent in programs written in imperative (and even functional) programming languages.

Be it execution, simulation or animation the basic idea is to provide inputs to the program, model or specification and observe the produced outputs or effects. Besides, we will show that $\{log\}$ offers more possibilities beyond this basic idea.

3.1 Basic simulations

Let's see an example of a simulation on a $\{log\}$ prototype. Assume the prototype of the birthday book is saved in a file named `bb.slog`. We start by executing the Prolog interpreter from a command terminal and from the folder where $\{log\}$ was installed[4].

```
~/setlog$ prolog

?- consult('setlog.slog').

?- setlog.

{log}=> consult('bb.slog').

{log}=> birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).
K = {},
B = {},
```

[3]In fact, the translation process can be automated in many cases.

[4]The name of the Prolog executable may vary depending on the interpreter and the operating system. The example corresponds to a Ubuntu Linux machine and SWI-Prolog.

```
K_ = {maxi},
B_ = {[maxi,160367]}

Another solution?  (y/n) y
no
{log}=>
```

where in each line we're doing the following:

1. The Prolog interpreter is executed.
2. The $\{log\}$ interpreter is loaded.
3. The $\{log\}$ interpreter is accessed.
4. The birthday book prototype is loaded.
5. The simulation is run:

   `birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).`

   consisting of:

   - `birthdayBookInit` is called passing to it any two variables as arguments;
   - `addBirthday` is called passing to it in the first and second arguments the same variables used to call `birthdayBookInit`; as the third and fourth arguments two constants; and two new variables in the last two parameters.

   Observe that the simulation ends in a dot.
6. $\{log\}$ shows the result of the simulation.
7. $\{log\}$ asks if we want to see other solutions and we answer yes.
8. $\{log\}$ says there are no more solutions.

Let's see the simulation in detail:

```
birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).
```

When we call `birthdayBookInit(K,B)`, `K` and `B` unify with `Known` and `Birthday` which are the formal parameters used in the definition of `birthdayBookInit` (see the complete code in Appendix A). This implies that `K` is equal to `Known` and `B` is equal to `Birthday` which in turn implies that `K` and `B` are equal to `{}`. This is exactly the first line of the answer returned by $\{log\}$. Hence, when `addBirthday(K,B,maxi,160367,K_,B_)` is called, it's like we were calling:

```
addBirthday({},{},maxi,160367,K_,B_)
```

Calling `addBirthday` implies the *non-deterministic* invocation of `addBirthdayOk` and `nameAlreadyExists`. That is, both clauses are invoked in an unspecified order. Let's assume that `addBirthdayOk` is invoked first. In this case, unification goes as follows:

```
Known = {}
Birthday = {}
Name_i = maxi
Date_i = 160367
K_ = Known_
B_ = Birthday_
```

Hence `addBirthdayOk` is instantiated as follows:

```
maxi nin {} &
un({},{maxi},K_) &
un({},{[maxi,160367]},B_)
```

which reduces to:

```
K_ = {maxi} &
B_ = {[maxi,160367]}
```

which corresponds to the second line of the answer returned by $\{log\}$.

When 'y' is pressed $\{log\}$ invokes `nameAlreadyExists` because it was the clause pending of invocation. Again, unification takes place and a new series of equations are produced:

```
Known = {}
Birthday = {}
Name_i = maxi
K_ = Known
B_ = Birthday
```

which implies that `K` unifies with `{}`. Then, `nameAlreadyExists` is instantiated as follows:

```
maxi in {}
```

As this predicate is obviously false, the invocation to `nameAlreadyExists` fails and hence $\{log\}$ produces no solution. As a consequence $\{log\}$ answers `no` after we press 'y'.

The following simulation is longer and includes the previous one.

```
birthdayBookInit(K,B)                    & addBirthday(K,B,maxi,160367,K1,B1) &
addBirthday(K1,B1,'Yo',201166,K2,B2)     & findBirthday(K2,B2,'Yo',C,K3,B3) &
addBirthday(K3,B3,'Otro',201166,K4,B4)   & remind(K4,B4,160367,Card,K5,B5) &
remind(K5,B5,201166,Card1,K_,B_).
```

Here we can see that we're calling all the operations defined in the prototype; that we use different variables to chain the state transitions; and that it's possible to use constants beginning with an uppercase letter as long as we enclose them between single quotation marks.

The first solution returned by that simulation is the following:

```
K = {},
B = {},
K1 = {maxi},
B1 = {[maxi,160367]},
K2 = {maxi,Yo},
B2 = {[maxi,160367],[Yo,201166]},
C = 201166,
K3 = {maxi,Yo},
B3 = {[maxi,160367],[Yo,201166]},
K4 = {maxi,Yo,Otro},
B4 = {[maxi,160367],[Yo,201166],[Otro,201166]},
Card = {maxi},
K5 = {maxi,Yo,Otro},
B5 = {[maxi,160367],[Yo,201166],[Otro,201166]},
Card1 = {Yo,Otro},
K_ = {maxi,Yo,Otro},
B_ = {[maxi,160367],[Yo,201166],[Otro,201166]}
```

where we can see that $\{log\}$ gives us the chance to have a complete trace of the prototype execution. Note also that $\{log\}$ eliminates the single quotation marks we used to enclose some constants.

It's important to remark that the variables used to chain the state transitions (i.e. `K1`, `B1`, ..., `K5`, `B5`) must be all different. If done otherwise, the simulation might be incorrect. For instance:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K,B).
```

will fail as the values of `K` and `B` before invoking `addBirthday` can't unify with the values returned by it. In other words, the `K` and `B` as the first two arguments of `addBirthday` can't have the same value than the `K` and `B` used as the last two arguments. We could use the same variable for the before and after state of query state operations (for instance when we invoke `findBirthday` and `remid`).

So far the two simulations we have performed start in the initial state. It's quite simple to start a simulation from any state:

```
K = {maxi,caro,cami,alvaro} &
B = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
addBirthday(K,B,'Yo',160367,K1,B1) & remind(K1,B1,160367,Card,K1,B1).
```

where we can see that we use the same variable to indicate the before and after state of `remid` (because we know this clause produces no state change). In this case the answer is:

```
K = {maxi,caro,cami,alvaro},
B = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]},
K1 = {maxi,caro,cami,alvaro,Yo},
B1 = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400],[Yo,160367]},
Card = {maxi,Yo}
```

A potential problem of manually defining the initial state for a simulation is that this state, due to human error, might not verify the state invariant. Nevertheless, it's very easy to avoid this problem as we will see in Section 3.6.

3.2 Type checking and simulations

So far we haven't really used $\{log\}$'s typechecker. Actually when we consulted `bb.slog` the types weren't checked. In other words $\{log\}$ ignored the `dec_p_type` assertions included in `bb.slog`. This means that possible type errors weren't detected by $\{log\}$. In this sense $\{log\}$ executed all the simulations in untyped mode. In this section we'll see how to call the typechecker and how this affects simulations. Recall reading chapter 9 of $\{log\}$ user's manual for further details.

Type checking can be activated by means of the `type_check` command which should be issued before the file is consulted.

```
~/setlog$ prolog

?- consult('setlog.slog').

?- setlog.

{log}=> type_check.          % typechecker is active

{log}=> consult('bb.slog').
```

In this way, when $\{log\}$ executes command `consult` it invokes the typechecker and if there are type errors we'll see an error message.

Type checking can be deactivated at any time by means of command `notype_check`.

When the typechecker is active all formulas (or programs or simulations) must be correctly typed because otherwise $\{log\}$ will just print a type error.

```
{log}=> birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).

***ERROR***: type error: variable K has no type declaration
```

Then, we have to declare the type of all variables:

```
{log}=> birthdayBookInit(K,B) & addBirthday(K,B,name?maxi,date?d160367,K_,B_) &
        dec([K,K_],kn) & dec([B,B_],bb).

K = {},
B = {},
K_ = {name?maxi},
B_ = {[name?maxi,date?d160367]}
```

Observe that in this case we have written `date?d160367` instead of `date?160367` (without 'd') because in $\{log\}$ constants of basic type `t` are of the form `t?atom` where `atom` must be a Prolog atom (numbers aren't atoms).

If the user wants to typecheck the program, for instance `bb.slog`, but (s)he doesn't want to deal with types when running simulations, the typechecker can be deactivated right after consulting the program. In this way $\{log\}$ will check the types of the program but it then will accept untyped formulas (or programs or simulations).

Clearly, in general, working with untyped simulations is easier but more dangerous because we could call the program with ill-typed inputs thus causing false failures.

In the rest of this section we'll work with untyped simulations. This means that the user must ensure that typechecking is deactivated (command `notype_check`).

3.3 Simulaciones using integer numbers

As we have said, $\{log\}$ is, essentially, a set solver. However, it's also capable of solving formulas containing predicates over the integer numbers. In that regard, $\{log\}$ uses two external solvers known as CLP(FD)[5] and CLP(Q)[6]. Each of them has its advantages and disadvantages.

By default $\{log\}$ uses CLP(Q). Users can change to CLP(FD) by means of command `int_solv-er(clpfd)` and can come back to CLP(Q) by means of `int_solver(clpq)`.

Generally speaking, it's more convenient to run simulations when CLP(FD) is active because it tends to generate more concrete solutions. In particular CLP(FD) is capable of performing labeling over the integer numbers which allows users to go through the solutions interactively. Labeling works if at least some of the integer variables are bound to a finite domain. Variable `N` is bound to the finite domain `int(a,b)` (`a` and `b` integer numbers) if predicate `N in int(a,b)` is in the formula. See chapter 7 of $\{log\}$ user's manual for more details.

For example, if CLP(Q) is active, the answer to the following goal:

[5] https://www.swi-prolog.org/pldoc/man?section=clpfd-predicate-index
[6] https://www.swi-prolog.org/pldoc/man?section=clpqr

```
Turn is 2*N + 1.
```

is exactly the same formula. That is, $\{log\}$ is telling us that the formula is satisfiable but we don't have one of its solutions. If we activate CLP(FD):

```
int_solver(clpfd).

Turn is 2*N + 1.
```

$\{log\}$ prints a warning message and the same formula:

```
***WARNING***: non-finite domain

true
Constraint: Turn is 2*N+1
```

This means that the formula *might be* satisfiable but CLP(FD) isn't sure. If we want a more reliable answer we have to bound `Turn` or `N` to a finite domain:

```
N in int(1,5) & Turn is 2*N + 1.
```

in which case the first solution is:

```
N = 1, Turn = 3
```

and we can get more solutions interactively. On the contrary, if we activate CLP(Q) the finite domain doesn't quite help to get a concrete solution:

```
int_solver(clpq).

N in int(1,5) & Turn is 2*N + 1.

true
Constraint: N>=1, N=<5, Turn is 2*N+1
```

On the other hand, CLP(Q) is complete for linear integer arithmetic while CLP(FD) isn't. This means that if we want to use $\{log\}$ to *automatically prove* a property of the program *for all the integer numbers*, we must use CLP(Q)[7]. Given that simulations don't prove properties it's reasonable to use CLP(FD).

3.4 Symbolic simulations

The symbolic execution of a program means to execute it providing to it variables as inputs instead of constants. This means that the executing engine should be able to symbolically operate with variables in order to compute program states as the execution moves forward. As a symbolic execution operates with variables, it can show more general properties of the program than when this is run with constants as input.

$\{log\}$ is able to symbolically simulate prototypes, within certain limits. These limits are given by set theory and non-recursive clauses. The following are the conditions under which $\{log\}$ can run symbolic simulations[8]:

[7] As in general non-linear arithmetic is undecidable it's quite difficult to build a tool capable of automatically proving program properties involving non-linear arithmetic.

[8] This is an informal description and not entirely accurate of the conditions for $\{log\}$ being able to perform symbolic simulations. These conditions are more or less complex and quite technical.

1. Recursive clauses are not allowed (for instance `sum_da` shown in Section 2.10).
2. Only the operators of Tables 1 and 2 are allowed. If the $\{log\}$ program uses the cardinality operator (`size`), the program can't use the operators of Table 2. The `size` operator is complete only when combined with the operators of Table 1.
3. All the arithmetic formulas are linear[9].

This means the $\{log\}$ code can't use operators of Table 3 if symbolic simulations are to be done[10]. Actually, many symbolic simulations are still possible even if the above conditions aren't met.

The $\{log\}$ prototype of the birthday book falls within the limits of what $\{log\}$ can symbolically simulate. For example, starting from the initial state we can call `addBirthday` using just variables:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K_,B_).
```

in which case $\{log\}$ answers:

```
K = {},
B = {},
K_ = {N},
B_ = {[N,C]}
```

which is a representation of the expected results. Now we can chain a second invocation to `addBirthday` using other input variables:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K1,B1) & addBirthday(K1,B1,M,D,K_,B_).
```

in which case the first solution returned by $\{log\}$ is:

```
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K_ = {N,M},
B_ = {[N,C],[M,D]}
Constraint: N neq M
```

where we can see a section of the answer which has never appeared before. Indeed, the most general solution that can be returned by $\{log\}$ consists of two parts: a (possibly empty) list of equalities between variables and terms (or expressions); and a (possibly empty) list of *constraints*. Each constraint is a $\{log\}$ predicate; the returned constraints appear after the word `Constraint`. The conjunction of all these constraints is always satisfiable (in general the solution is obtained by substituting the variables of type set by the empty set). In this example, clearly, the second invocation to `addBirthday` can add the pair `[M,D]` to the birthday bool if and only if `M nin {N}` (i.e. $M \notin \{N\}$), which holds if and only if `M` is different from `N`.

$\{log\}$ returns a second solution to this symbolic execution:

```
K = {},
B = {},
K1 = {N},
```

[9] More precisely, all the integer expressions must be sums or subtractions of terms of the form `x*y` with `x` or `y` constants. All arithmetic relational operators are allowed, even `neq`.

[10] The problem with the operators of Table 3 is that they depend on certain aspects of set theory that aren't implemented in $\{log\}$, yet.

```
B1 = {[N,C]},
M = N,
K_ = {N},
B_ = {[N,C]}
```

produced after considering that `N` and `M` are equal in which case the second invocation to `addBirthday` goes through the branch of `nameAlreadyExists` and so `K_` and `B_` are equal to `K1` and `B1`, which is the expected result as well.

Clearly, symbolic executions allows us to draw more general conclusions about the behavior of the prototype. The next example illustrates what we just said:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K1,B1) &
addBirthday(K1,B1,M,D,K2,B2) & findBirthday(K2,B2,W,X,K2,B2).
```

given that $\{log\}$ will consider several particular cases regarding whether `M`, `N` and `W` are equal or not. For example, the following are the first three solutions returned by $\{log\}$:

```
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K2 = {N,M},
B2 = {[N,C],[M,D]},
W = N,
X = C
Constraint: N neq M

Another solution?  (y/n)
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K2 = {N,M},
B2 = {[N,C],[M,D]},
W = M,
X = D
Constraint: N neq M

Another solution?  (y/n)
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K2 = {N,M},
B2 = {[N,C],[M,D]}
Constraint: N neq M, N neq W, M neq W
```

In the first case `W = N` is considered and so `X` must be equal to `C`; the second case is similar to the first one; and in the third `W` isn't `M` nor `N` and so `X` can take any value (is this an error?). $\{log\}$ returns more solutions some of which are repeated.

Obviously symbolic simulations may combine variables with constants. In general the less the variables we use the less the number of solutions.

3.5 Inverse simulations

Normally, in a simulation the user provides inputs and the model returns the outputs. There are situations in which is interesting to get the inputs from the outputs. This means a sort of an inverse simulation.

$\{log\}$ is able to perform inverse simulations within the same limits in which is able to perform symbolic simulations. In fact, a careful reading of the previous section reveals that $\{log\}$ doesn't really distinguish input from output variables, nor between before and after states. As a consequence, for $\{log\}$ is more or less the same to simulate a prototype by providing values for the input variables or for the output variables; in fact, $\{log\}$ is able to simulate a prototype just with variables.

Let's see a very simple inverse simulation where we only give the after state:

```
K_ = {maxi,caro,cami,alvaro} &
B_ = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
addBirthday(K,B,N,C,K_,B_).
```

The first solution returned by $\{log\}$ is the following:

```
K_ = {maxi,caro,cami,alvaro},
B_ = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]},
K = {maxi,caro,cami},
B = {[maxi,160367],[caro,201166],[cami,290697]},
N = alvaro,
C = 110400
```

When the Z specification is deterministic, the corresponding $\{log\}$ prototype will be deterministic as well. Therefore, for any given input there will be only one solution. However, the inverse simulation of a deterministic prototype may generate a number of solutions. This is the case with the above simulation. The first solution computed by $\{log\}$ considered the case where `N = alvaro` and `C = 110400`, but this isn't the only possibility. Then, when we ask $\{log\}$ for another solution we get, for instance, the following:

```
K_ = {maxi,caro,cami,alvaro},
B_ = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]},
K = {maxi,caro,alvaro},
B = {[maxi,160367],[caro,201166],[alvaro,110400]},
N = cami,
C = 290697
```

which means that `K_` and `B_` may have been generated by starting from some `K` and `B` where `cami`'s birthday isn't in the book and so we can add it.

3.6 Evaluation of predicates

At the end of Section 3.1 we showed how to start a simulation from a state different from the initial state. We also said that this entails some risks as manually writing the start state is error prone which may lead to an unsound state. In this section we will see how to avoid this problem by using a feature of $\{log\}$ that is useful for other verification activities, too.

Let's consider the following state of the birthday book:

```
Known = {maxi,caro,cami,alvaro}
Birthday = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]}
```

Starting a simulation from this state may give incorrect results if it doesn't verify the state invariant defined for the specification. Recall that the state invariant for the birthday book is:

BirthdayBookInv
BirthdayBook

$known = \operatorname{dom} birthday$

With $\{log\}$ is very simple to check that `S0` verifies the state invariant. We first translate *BirthdayBookInv* into $\{log\}$:

```
birthdayBookInv(Known,Birthday) :-
  dom(Birthday,Known).
```

Afterwards, we invoke the new clause passing in to it the state defined above:

```
Known = {maxi,caro,cami,alvaro} &
Birthday = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
birthdayBookInv(Known,Birthday).
```

in which case $\{log\}$ returns the values of `Known` and `Birthday`, meaning that `birthdayBookInv` is satisfied. If this weren't the case the answer would have been `no`. If we call `birthdayBookInv` passing in to it the values of the variables:

```
birthdayBookInv({maxi,caro,cami,alvaro},
                {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]}).
```

the answer is `yes`.

Now, let's say we have written a state not satisfying the state invariant, for instance (note that `maxi` is missing from `known`):

```
Known = {caro,cami,alvaro} &
Birthday = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
birthdayBookInv(Known,Birthday).
```

Then, the answer of $\{log\}$ is `no`.

In general we can take any property and evaluate it on some values. For example, `Birthday` must be a partial function in any given state. Hence, for instance, if we execute:

```
pfun({[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]}).
```

$\{log\}$ answers `yes` and if we execute:

```
pfun({[maxi,160367],[maxi,201166],[cami,290697],[alvaro,110400]}).
```

it answers `no`.

4 Automated proofs with $\{log\}$

Evaluating properties with $\{log\}$ helps to run correct simulations by checking that the starting state is correctly defined. It also helps to *test* whether or not certain properties are true of the specification or not. However, it would be better if we could *prove* that these properties are true of the specification. In this section we will see how $\{log\}$ allows us to prove that the operations of a specification preserve the state invariant.

So far we have used $\{log\}$ as a (prototype) programming language. However, $\{log\}$ is also a *satisfiability solver*. This means that $\{log\}$ is a program that can decide if formulas of some theory are *satisfiable* or not. In this case the theory is the theory of finite sets and binary relations combined with the operators listed in Tables 1 and 2 combined with the linear integer arithmetic[11].

If F is a formula depending on a variable, we say that F is *satisfiable* if and only if:

$$\exists y : F(y)$$

In the case of $\{log\}$, y is quantified over *all* finite sets. Therefore, if $\{log\}$ answers that F is satisfiable it means that there exists a finite set satisfying it. Symmetrically, if $\{log\}$ says that F isn't satisfiable it means that there is no finite set satisfying it. Formally, F is an unsatisfiable formula if:

$$\forall y : \neg F(y) \tag{1}$$

If we call $G(x) \mathrel{\widehat{=}} \neg F(x)$ then (1) becomes:

$$\forall y : G(y) \tag{2}$$

which means that G is true of every finite set. Putting it in another way, G is *valid* with respect to the theory of finite sets; or, equivalently, G is a *theorem* of the theory of finite sets.

In summary, if $\{log\}$ decides that F *is unsatisfiable*, then we know that $\neg F$ *is a theorem*.

In other words, (1) and (2) are two sides of the same coin: (1) says that F is unsatisfiable and (2) says that G (i.e. $\neg F$) is a theorem.

Recall to activate CLP(Q) when performing proofs.

Recall that in order to prove that an operation, T, of a Z specification preserves the state invariant I we have to discharge the following proof obligation:

$$I \wedge T \Rightarrow I' \tag{3}$$

If we want to use $\{log\}$ to discharge (3) we have to ask $\{log\}$ to check if the negation of (3) is *unsatisfiable*. In fact, we need to execute the following $\{log\}$ *program*:

$$I \wedge T \wedge \neg I' \tag{4}$$

because $\neg (I \wedge T \Rightarrow I') \equiv \neg (\neg (I \wedge T) \vee I') \equiv I \wedge T \wedge \neg I'$.

The problem in using $\{log\}$ to decide the satisfiability of (4), is the predicate $\neg I'$. More precisely, we shouldn't use logical negation because if we do, in general, $\{log\}$ won't be able to prove that the formula is unsatisfiable. In any case, logical negation can be avoided in $\{log\}$ *as long as we work with the operators of Tables 1 and 2 and linear integer algebra*. Recall that for each operator of Tables 1 and 2 $\{log\}$ implements its negation. For example, instead of $\neg x \in A$ we should write $x \notin A$; instead of $\neg A = B$, we should write $A \neq B$; instead of $\neg pfun(f)$ we should write $npfun(f)$; instead of $\neg X > Y$

[11]In what follows we will only mention the theory of finite sets but the same is valid for this theory combined with linear integer algebra.

we should write $X \leq Y$; and so forth and so on. Then, coming back to formula (4), if I' is a compound predicate (i.e. a predicate formed by conjunction, disjunction, implication, etc.), then we must distribute the negation all the way down to the atoms at which point we use the negations of the operators of Tables 1 and 2.

In the Z specification of the birthday book we proposed as invariant the following formula:

$$\operatorname{dom} birthday = known$$

Hence, if we want to prove that *AddBirthday* preserves this invariant, we have to prove the following:

$$\operatorname{dom} birthday = known \land AddBirthday \Rightarrow \operatorname{dom} birthday' = known' \tag{5}$$

However, it we're going to use $\{log\}$ to prove (5) we need to see if the translation into $\{log\}$ of the following formula is unsatisfiable (due to (4)):

$$\operatorname{dom} birthday = known \land AddBirthday \land \operatorname{dom} birthday' \neq known'$$

The translation into $\{log\}$ of that formula is:

```
dom(B,K) &
addBirthday(K,B,N,C,K_,B_) &
ndom(B_,K_).
```

When $\{log\}$ is called to solve this formula (or to run this program) the answer is `no` because $\{log\}$ finds it unsatisfiable. If this formula is unsatisfiable, then its negation is a theorem. And the negation of this $\{log\}$ formula is (5). Hence, if $\{log\}$ answers `no` we know that *AddBirthday* preserves the state invariant.

So we have automatically proved that *AddBirthday* preserves this invariant for every *finite* set. From this, we can't conclude that *AddBirthday* preserves this invariant for every set but, will the implementation of the birthday book ever process an *infinite* set of persons? The reader can try to prove that the other operations of the birthday book preserve this invariant.

When the Z specification of the birthday book was translated into $\{log\}$, we mentioned that when translating *birthday* its type at the $\{log\}$ level is `stype([name,date])`—which is interpreted as the Z type $NAME \leftrightarrow DATE$. Besides we said we didn't encode *birthday* as a function because in $\{log\}$ functions aren't a type (as they are in, for instance, functional programming languages). Finally, we said that we will use $\{log\}$ to prove that *birthday* is a function in every state, i.e. it's a state invariant. In other words, with $\{log\}$'s type system we can express that *birthday* is a binary relation between *NAME* and *DATE*; and then we can use $\{log\}$ to prove that actually *birthday* is a function. For more details reread Section 2.1.2; in particular we suggest to take a look at shcema *BirthdayBookInv*.

Then, we will prove that:

$$birthday \in \_ \nrightarrow \_ \land AddBirthday \Rightarrow birthday' \in \_ \nrightarrow \_ \tag{6}$$

where $\_ \nrightarrow \_$ denotes the set of all partial functions (with any domain and range). Again, as we will use $\{log\}$ to prove that, we need to negate the formula:

$$birthday \in \_ \nrightarrow \_ \land AddBirthday \land birthday' \notin \_ \nrightarrow \_$$

where we interpret $birthday' \notin \_ \nrightarrow \_$ as $birthday'$ isn't a partial function. Now, the corresponding $\{log\}$ formula is:

```
pfun(B) &
addBirthday(K,B,N,C,K_,B_) &
npfun(B_).
```

where we translate $birthday' \notin \_ \rightarrowtail \_$ as `npfun(B_)` because this predicate states that its argument isn't a partial function (see Table 2). When $\{log\}$ is asked to solve the above formula (or to run the above program) the answer is the following:

```
B = {[N,_N2]/_N1},
K_ = {N/K},
B_ = {[N,_N2],[N,C]/_N1}
Constraint: pfun(_N1), comppf({[N,N]},_N1,{}), N nin K, set(_N1), [N,C] nin _N1,
            C neq _N2, [N,_N2]nin _N1, set(K)
```

What? Weren't we expecting `no` *for an answer? The whole idea is that AddBirthday does preserve the fact that birthday is a function, isn't it? Then, what does it mean that $\{log\}$ returned a solution?* Clearly, it means that (6) isn't a theorem. Why (6) isn't a theorem (we were pretty sure of that)? The answer returned by $\{log\}$ helps us to understand the problem. Take a look at it and observe that `B = {[N,_N2]/_N1}` but it doesn't say that `N` belongs to `K` when, as we proved in (5), `K` is equal to the domain of `B` (and `N` clearly belongs to the domain of `B`). The problem is that *AddBirthday* states that $n? \notin known$ instead of $n? \notin \operatorname{dom} birthday$. As we know that $known = \operatorname{dom} birthday$ then $n? \notin known$ and $n? \notin \operatorname{dom} birthday$ are equivalent. But $known = \operatorname{dom} birthday$ is not part of (6), and so $\{log\}$ can't use this equality in the proof. Given that we have *already* proved that $known = \operatorname{dom} birthday$ is a state invariant we can *assume* it holds in (6). Then, we now call $\{log\}$ to solve the following formula:

```
dom(B,K) &                              %%% hypothesis
pfun(B) &
addBirthday(K,B,N,C,K_,B_) &
npfun(B_).
```

and now $\{log\}$ answers `no`.

Recall that we don't need to prove that the domain of *birthday* is a subset of *NAME* and that its range is a subset of *DATE* because this is guaranteed by the typechecker.

In addition of proving that each operation of the specification *preserves* the state invariant we must prove that the initial state *satisfies* it (because otherwise the system would start from an unsound state and thus the whole system would be at danger). In general, this proof is simpler. We start by proving that in the initial state $\operatorname{dom} birthday = known$ is satisfied:

```
birthdayBookInit(K,B) &
ndom(B,K).
```

We follow by proving that in the initial state *birthday* is a partial function:

```
birthdayBookInit(K,B) &
npfun(B).
```

And we finish by proving that the initial domain of *birthday* is a subset of *NAME*:

```
birthdayBookInit(K,B) &
dom(B,DB) & ran(B,RB) &
(nsubset(DB,NAME) or nsubset(RB,DATE)).
```

4.1 Typechecking and proofs

As with simulations, proofs can be done with or without the typechecker activated. All we did above assumes the typechecker isn't active (command `notype_check`). Activating the typechecker makes proofs somewhat more cumbersome (as users need to give the type of all variables) but they are safer (because the typechecker can catch errors that $\{log\}$'s deductive system can't).

Given that from a verification perspective proofs are more critical than simulations, it is recommended to activate the typechecker before discharging the invariance lemmas.

We will show, then, how to run proofs when the typechecker is active (command `type_check`). If we run the proof without declaring the types of variables:

```
dom(B,K) &
addBirthday(K,B,N,C,K_,B_) &
ndom(B_,K_).
```

we get a type error:

```
***ERROR***: type error: variable B has no type declaration
```

Hence, we have to give the type for each variable by means of predicate `dec`:

```
dec([B,B_],bb) & dec([K,K_],kn) & dec(N,name) & dec(C,date) &
dom(B,K) &
addBirthday(K,B,N,C,K_,B_) &
ndom(B_,K_).
```

in which case we get a `no` answer. Clearly, if types are wrong:

```
dec([B,B_],bb) & dec([K,K_],kn) & dec(N,date) & dec(C,name) &
dom(B,K) &
addBirthday(K,B,N,C,K_,B_) &
ndom(B_,K_).
```

we get a type error:

```
***ERROR***: type error:
  in addBirthday(K,B,N,C,K_,B_) arguments have the wrong type:
    N is date but should be name
    C is name but should be date
```

References

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

A The $\{log\}$ implementation of the birthday book

```
:- dec_type(bb,stype([name,date])).
:- dec_type(kn,stype(name)).

:- dec_p_type(birthdayBookInit(kn,bb)).
birthdayBookInit(Known,Birthday) :-
```

```
  Known = {} &
  Birthday = {}.

:- dec_p_type(birthdayBookInv(kn,bb)).
birthdayBookInv(Known,Birthday) :-
  dom(Birthday,Known).

:- dec_p_type(addBirthdayOk(kn,bb,name,date,kn,bb)).
addBirthdayOk(Known,Birthday,Name_i,Date_i,Known_,Birthday_) :-
  Name_i nin Known &
  un(Known,{Name_i},Known_) &
  un(Birthday,{[Name_i,Date_i]},Birthday_).

:- dec_p_type(nameAlreadyExists(kn,bb,name,kn,bb)).
nameAlreadyExists(Known,Birthday,Name_i,Known_,Birthday_) :-
  Name_i in Known &
  Known_ = Known &
  Birthday_ = Birthday.

:- dec_p_type(addBirthday(kn,bb,name,date,kn,bb)).
addBirthday(Known,Birthday,Name_i,Date_i,Known_,Birthday_) :-
  addBirthdayOk(Known,Birthday,Name_i,Date_i,Known_,Birthday_)
  or
  nameAlreadyExists(Known,Birthday,Name_i,Known_,Birthday_).

:- dec_p_type(findBirthdayOk(kn,bb,name,date,kn,bb)).
findBirthdayOk(Known,Birthday,Name_i,Date_o,Known_,Birthday_) :-
  Name_i in Known &
  apply(Birthday,Name_i,Date_o) &
  Known_ = Known &
  Birthday_ = Birthday.

:- dec_p_type(notAFriend(kn,bb,name,kn,bb)).
notAFriend(Known,Birthday,Name_i,Known_,Birthday_) :-
  Name_i nin Known &
  Known_ = Known &
  Birthday_ = Birthday.

:- dec_p_type(findBirthday(kn,bb,name,date,kn,bb)).
findBirthday(Known,Birthday,Name_i,Date_o,Known_,Birthday_) :-
  findBirthdayOk(Known,Birthday,Name_i,Date_o,Known_,Birthday_)
  or
  notAFriend(Known,Birthday,Name_i,Known_,Birthday_).

:- dec_p_type(remind(kn,bb,date,kn,kn,bb)).
remind(Known,Birthday,Today_i,Cards_o,Known_,Birthday_) :-
```

```
rres(Birthday,{Today_i},M) & dec(M,bb) &
dom(M,Cards_o) &
Known_ = Known &
Birthday_ = Birthday.
```