From B Specifications to $\{log\}$ Forgrams

Maximiliano Cristiá and Gianfranco Rossi

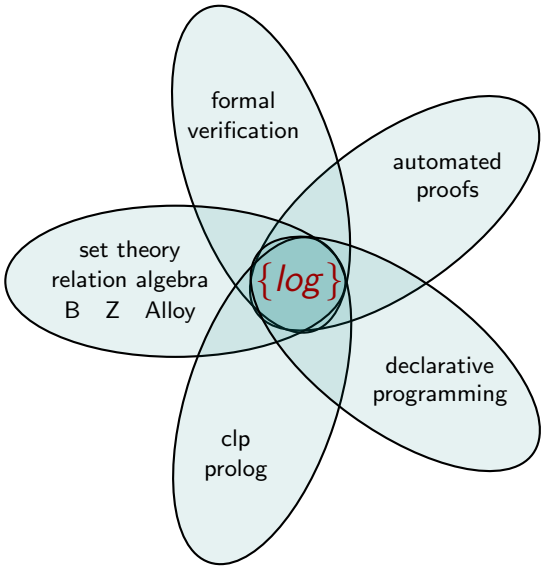
Contents

1 What is {*log*}?

{*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*} in different directions.

As shown below, {*log*} is at the intersection of several Computer Science areas. {*log*} can be used as a *formal verification* tool because it performs *automated proofs* over a very expressive theory. It's also a *declarative programming language* meaning that programmers have to expresses the logic of a computation without describing its control flow. In particular, {*log*} implements declarative programming as an instance of a *constraint logic programming* (CLP) system implemented in Prolog. The code written in {*log*} is quite similar (in its essence, not in its form) to formal specifications written in languages based on set theory and set relation algebra such as B, Z and Alloy.



1.1 Installation

{*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*}, all the library files and its user's manual from here:

`https://www.clpset.unipr.it/setlog.Home.html`

You should also read {*log*} user's manual:

`https://www.clpset.unipr.it/SETLOG/setlog-man.pdf`

1.2 Using {*log*}

As we have said, {*log*} is a *satisfiability solver*. This means that {*log*} is a program that determines whether or not a given formula is satisfiable. Once yo access {*log*} it presents a prompt:

```
{log}=>
```

You can now ask {*log*} to solve formulas. For example:

```
{log}=> un({a,2},B,{X,2,c}).
```

The atomic predicate `un({a,2},B,{X,2,c})` means $\{a,2\} \cup B = \{X,2,c\}$, where X and B are variables and a and c are constants. In {*log*} variables begin with a uppercase letter, and constants begin with lowercase letters. Note that the formula ends with a dot. Hence, when we type in that formula {*log*} will try to find values for `B` and `X` that satisfy the formula—this is why we say that {*log*} is a satisfiability solver. So, {*log*} asks itself, are there values for `B` and `X` that make the formula true? {*log*} answers the following:

```
B = {c},
X = a

Another solution?  (y/n)
```

As you can see, {*log*} produces a solution and asks whether or not we want to see other solutions. In this case there are three more solutions:

```
B = {2,c},
X = a

Another solution?  (y/n)
B = {a,c},
X = a

Another solution?  (y/n)
B = {a,2,c},
X = a

Another solution?  (y/n)
no

{log}=>
```

When there are no more solutions or when we don't type in '`y`', {*log*} says '`no`' and prints the prompt again.

Let's try another example.

```
{log}=> un({a,2},B,{X,2,c}) & c nin B.
```

The atomic predicate `c nin B` means $c \notin B$ and '`&`' means conjunction ($\wedge$). In this case {*log*} answers `no`. Why is that? Because there are no values for `B` and `X` that make the formula true. Clearly, as `c` doesn't belong to `{a,2}` but at the same time it belongs to the union between that set

and `B` the only chance to satisfy the formula is when `c` belongs to `B`. But we rule this possibility out by conjoining `c nin B`. Then, {*log*} is saying "your formula is unsatisfiable".

Summarizing, if we see anything different from '`no`' we know the formula is satisfiable; otherwise, it's unsatisfiable.

2 An example of a B specification translated into {*log*}

These class notes are focused in showing how B specifications can be translated into {*log*} and, later, on how {*log*} can be used to run simulations and automated proofs.

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

We have already learned to write some B specifications. Here, we will show how these B specifications can be translated into {*log*}. To that end we will use a running example. Later on we will explain with some detail how B elements not appearing in the example can be translated into {*log*}; we will see that some B elements can be translated in more than one way.

2.1 The running example

The specification to be used as running example is 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. The problem is borrowed from [1].

2.2 The B specification

The B machine containing the specification of the birthday book system will be called *BirthdayBook*. In our account of the system, we need to deal with people's names and with dates. We also need a type for the messages outputted by some of the operations. Then, we introduce the following types.

MACHINE *BirthdayBook*
SETS *NAME*; *DATE*; *MSG* = {*ok*, *nameExists*}
. . . . . .
END

Now, we define two state variables for our machine:

MACHINE *BirthdayBook*
SETS *NAME*; *DATE*; *MSG* = {*ok*, *nameExists*}
VARIABLES *known*, *birthday*
. . . . . .
END

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 invariant of our machine is the following.

MACHINE *BirthdayBook*
SETS $NAME$; $DATE$; $MSG = \{ok, nameExists\}$
VARIABLES $known, birthday$
INVARIANT $known \in \mathbb{P}\, NAME \land birthday \in NAME \pfun DATE \land known = \mathrm{dom}(birthday)$
. . . . . .
END

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. Besides the types for the variables are in accordance with the intended use described above.

The initial state of the birthday book is the following.

MACHINE *BirthdayBook*
SETS $NAME$; $DATE$; $MSG = \{ok, nameExists\}$
VARIABLES $known, birthday$
INVARIANT $known \in \mathbb{P}\, NAME \land birthday \in NAME \pfun DATE \land known = \mathrm{dom}(birthday)$
INITIALIZATION $known, birthday := \{\}, \{\}$
. . . . . .
END

The first operation we specify is how to add a birthday to the birthday book. As we did with the savings account specification we model the normal and abnormal behaviors outputting convenient messages in each case.

$msg \leftarrow$ **addBirthday**$(name, date) \mathrel{\widehat{=}}$
  PRE $name \in NAME \land date \in DATE$
  THEN
    IF $name \notin known$
    THEN $known, birthday, msg := known \cup \{name\}, birthday \cup \{name \mapsto date\}, ok$
    ELSE $msg := nameExists$
    END
  END

Note how both state variables are updated accordingly.

MACHINE *BirthdayBook*
SETS *NAME*; *DATE*; *MSG* = {*ok*, *nameExists*}
VARIABLES *known*, *birthday*
INVARIANT $known \in \mathbb{P} NAME \land birthday \in NAME \pfun DATE \land known = \mathrm{dom}(birthday)$
INITIALIZATION $known, birthday := \{\}, \{\}$
OPERATIONS
$msg \leftarrow$ **addBirthday**$(name, date) \mathrel{\widehat{=}}$
PRE $name \in NAME \land date \in DATE$
THEN
IF $name \notin known$
THEN $known, birthday, msg := known \cup \{name\}, birthday \cup \{name \mapsto date\}, ok$
ELSE $msg := nameExists$
END
END;
$date \leftarrow$ **findBirthday**$(name) \mathrel{\widehat{=}}$
PRE $name \in NAME \land name \in known$
THEN $date := birthday(name)$
END;
$cards \leftarrow$ **remind**$(today) \mathrel{\widehat{=}}$
PRE $today \in DATE$
THEN $cards := \mathrm{dom}(birthday \rhd \{today\})$
END
END

Figure 1: B specification of the birthday book

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

$date \leftarrow$ **findBirthday**$(name) \mathrel{\widehat{=}}$
PRE $name \in NAME \land name \in known$
THEN $date := birthday(name)$
END

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

$cards \leftarrow$ **remind**$(today) \mathrel{\widehat{=}}$
PRE $today \in DATE$
THEN $cards := \mathrm{dom}(birthday \rhd \{today\})$
END

The complete B specification of the birthday book can be seen in Figure 1.

2.3 The {*log*} forgram

The {*log*} forgram resulting from the translation of the B specification must be saved in a file with extension `.pl` or `.slog`. It is convenient to put this file in the same folder where {*log*} was installed.

A B machine is translated as a collection of {*log*} *clauses* and *declarations* written in a single file. 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. 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. Then, for instance, the state variables of the birthday book will be `Known` and `Birthday`, instead of `known` and `birthday` because in this case they would be constants. We'll see how variables are typed in Section 3. For now we'll not pay much attention to types.

2.3.1 Translating the Sets section

In general, the Sets sections is not translated into {*log*}. The sets declared in this section can be freely introduced in {*log*}. We'll see more on this in Section 3.

2.3.2 Translating the Variables section

The Variables section is translated as a {*log*} declaration as follows:

```
variables([Known, Birthday]).
```

Note that declarations end with a dot ('`.`').

2.3.3 Translating the Invariant section

Before translating the invariant we normalize it:

$$\begin{aligned}\textsc{Invariant}\ & known \in \mathbb{P}\,NAME \wedge birthday \in NAME \leftrightarrow DATE \\ & \wedge\ pfun(birthday) \wedge known = \mathrm{dom}(birthday)\end{aligned}$$

The first part of the invariant ($known \in \mathbb{P}\,NAME \wedge birthday \in NAME \leftrightarrow DATE$) is translated as type declarations, whereas the second part is translated as a clause declared as invariant. Type declarations will be introduced in Section 3. The {*log*} code is the following:

```
invariant(birthdayBookInv).
birthdayBookInv(Known,Birthday) :- dom(Birthday,Known) & pfun(Birthday).
```

Then, the first line declares the clause named `birthdayBookInv` to be an invariant. The second line is a clause.

Clauses are of the form:

```
head(params) :- body.
```

where `body` is a {*log*} formula. In this case the formula is simply `dom(Birthday,Known) & pfun(Birthday)` which is equivalent to $known = \text{dom}(birthday) \wedge pfun(birthday)$.

Alternatively, you can split the invariant in smaller pieces. Actually, each conjunct in the INVARIANT section may become an invariant. This strategy is a good option when the specification is large and complex because later it will be easier for {*log*} to discharge invariance lemmas. In this case the {*log*} code look like this:

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

invariant(pfunInv).
pfunInv(Birthday) :- pfun(Birthday).
```

Note that declarations and clauses end with a dot ('`.`').

2.3.4 Translating the INITIALIZATION section

The INITIALIZATION section is translated as a declaration and a clause as follows:

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

That is, we first declare that the clause `birthdayBookInit` corresponds to the initial state of the system and then the clause is defined. Here there's an important difference w.r.t. the B specification because the body of the clause is a formula and not a multiple assignment. Indeed, `Known = {}` and `Birthday = {}` are predicates. We could have written them also as `{} = Known` and `{} = Birthday` because the symbol '=' is simply logical equality. In turn '`&`' means conjunction ($\wedge$). Hence, we could have written `birthdayBookInit` as follows:

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

In any case, the {*log*} implementation of the INITIALIZATION section follows the semantics of the B specification.

2.3.5 Translating operations

A B operation is translated as a clause and a declaration indicating that the clause is an operation. When a B operation is translated, the corresponding clause receives as arguments all the state variables, all the input parameters and all the output parameters. Besides, for each state variable v the clause will also receive $v\_$, which represents the value of v in the next state. That is, in {*log*} we have to represent the next state explicitly with a second set of variables. Hence, the head of the {*log*} clause corresponding to the B operation named **addBirthday** is the following:

```
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg)
```

where `Name` and `Date` correspond to input parameters *name* and *date* declared in **addBirthday**; `Known` and `Birthday` represent the before state while `Known_` and `Birthday_` represent the after state; and `Msg` corresponds to the output parameter.

Now we give the complete specification of the clause preceded by its declarion:

```
operation(addBirthday).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) :-
  (Name nin Known &
   un(Known,{Name},Known_) &
   un(Birthday,{[Name,Date]},Birthday_) &
   Msg = ok
  or
   Name in Known &
   Known_ = Known &
   Birthday_ = Birthday &
   Msg = nameExists
  ).
```

That is, the first line declares that `addBirthday` is an operation. Then, the **IF-THEN-ELSE** statement in **addBirthday** is translated as a logical disjunction ('`or`'). The condition of the conditional statement, $name \notin known$, is translated as `Name nin Known`. The word '`nin`' in $\{log\}$ means $\notin$. If the condition is true the **THEN** branch specifies the multi assignment:

$$known, birthday, msg := known \cup \{name\}, birthday \cup \{name \mapsto date\}, ok$$

This multi assignment is translated as a conjunction of $\{log\}$ constraints:

```
un(Known,{Name},Known_) & un(Birthday,{[Name,Date]},Birthday_) & Msg = ok
```

The meaning of these constraints is as follows:

- `un(Known,{Name_i},Known_)` means $Known\_ = Known \cup \{Name\}$.
  That is, in $\{log\}$ `un(A,B,C)` is equivalent to $C = A \cup B$.
- Similarly, `un(Birthday,{[Name,Date]},Birthday_)` is $Birthday\_ = Birthday \cup \{Name \mapsto Date\}$.
  That is, in $\{log\}$ the ordered pair $x \mapsto y$ is written as `[x,y]`.

When the condition of the **IF-THEN-ELSE** statement is false, we have the assignment $msg := nameExists$. This means that the state of the machine doesn't change and that the machine outputs *nameExists*. In $\{log\}$ we first need to write the negation of the condition, that is `Name in Known` or `neg(Name nin Known)`. Then, we must say that the machine doesn't change the state and that *nameExists* is outputted. We do this with the conjunction:

```
Known_ = Known & Birthday_ = Birthday & Msg = nameExists
```

As `Known_` and `Birthday_` represent the next state, the equalities `Known_ = Known` and `Birthday_ = Birthday` mean that the state doesn't change.

Finally, observe that the **PRE** section hasn't been translated. In this case the **PRE** section contains only type declarations ($name \in NAME \wedge date \in DATE$). The translation of type declarations will be seen in Section 3.

Now we give the translation of **findBirthday**.

```
operation(findBirthday).
findBirthday(Known,Birthday,Name,Date,Known,Birthday) :-
  Name in Known & applyTo(Birthday,Name,Date).
```

where `applyTo` is a predicate implementing function application. That is, `applyTo(F,X,Y)` is true if and only if $F(X) = Y$ holds. Note that `applyTo(F,X,Y)` makes sense only if `X` is in the domain of `F`, which in turn is a function *at least on* `X`. As with **addBirthday** the type declaration $name \in NAME$ isn't included in the body of the clause. Besides, note how we say that the operation doesn't change the state. Instead of including `Known_ = Known & Birthday_ = Birthday` in the body of the clause we don't include `Known_` and `Birthday_` in the head but two copies of the before-state variables. This is interpreted by {*log*} as the operation not changing the state. We couldn't do this in **addBirthday** because there's one branch of that operation that changes the state.

Finally, the translation of **remind** is the following:

```
operation(remind).
remind(Known,Birthday,Today,Cards,Known,Birthday) :-
  rres(Birthday,{Today},M) & dom(M,Cards).
```

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 $S = R \rhd A$. Then, the body of the clause corresponds to the following B predicate: $m = birthday \rhd \{today\} \land cards = dom(m)$. As **remind** doesn't change the state we repeat the state variables in the head of the clause.

3 Types in {*log*}

So far we haven't given the types of the variables. {*log*} provides a typechecker that can be activated and deactivated by the user. {*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 for the birthday book:

```
def_type(bb,rel(name,date)).
def_type(kn,set(name)).
def_type(msg,enum([ok,nameExists])).
```

where `bb` is a type identifier o synonym of the type `rel(name,date)`. In `rel(name,date)`, `name` and `date` correspond to the basic types *NAME* and *DATE* of the B specification. B basic types can be introduced in {*log*} without any previous declaration. In {*log*} basic types must begin with a lowercase letter (i.e. they are constants). In turn, `rel(name,date)` corresponds to the type of all binary relations between `name` and `date`. That is, `rel(name,date)` corresponds to $NAME \leftrightarrow DATE$ in B. `set(name)` corresponds to $\mathbb{P}NAME$ in B and `enum([ok,nameExists])` corresponds to the set $\{ok, nameExists\}$ which we named *MSG* in the B specification.

These type synonyms allow us to declare the type of the `addBirthday` operation:

```
dec_p_type(addBirthday(kn,bb,name,date,kn,bb,msg)).
```

The type declaration must come before the clause definition:

```
operation(addBirthday).
dec_p_type(addBirthday(kn,bb,name,date,kn,bb,msg)).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) :-
  (Name nin Known &
  ...
```

The `dec_p_type` declaration 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. Then, the type of `Known` is `kn`, the type of `Birthday` is `bb`, etc.

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

```
operation(remind).
dec_p_type(remind(kn,bb,date,kn,kn,bb)).
remind(Known,Birthday,Today,Cards,Known,Birthday) :-
  rres(Birthday,{Today},M) & dom(M,Cards) & dec(M,bb).
```

This clause is interesting because it shows how variables local to the clause 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*} forgram including type declarations of the complete translation of the birthday book can be found in Appendix A. As can be seen in that appendix, all the clauses, including `invariant` and `initial`, are typed.

Recall that partial functions *aren't* a type in B. 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 B we have $f \in X \pfun Y$ in {*log*} we declare `F` to be of type `rel(x,y)` and then we should prove that `F` is a function as an invariant. Likewise, if in B we declare $x \in \mathbb{N}$, in {*log*} we must declare `X` to be of type `int` and then prove that `0 =< X` is an invariant. In general, when a B specification is translated into {*log*} it is convenient to first normalize the B specification and then start the translation into {*log*}. In this case B 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, $x \in \mathbb{N}$ is a non-normalized declaration because $\mathbb{N}$ isn't a type (it's a set). The normalized declaration is $x \in \mathbb{Z}$ plus $x \geq 0$ conjoined in the INVARIANT section or in the **PRE** section of an operation. In this case, in {*log*} the type of x is `int` and we should prove that x is always greater than or equal to zero (i.e., that $0 \leq x$ is an invariant), or simply assert that as a precondition.

4 Translating B specifications into {*log*}

In this section we show how the most used elements appearing in B specification are translated into {*log*}.

4.1 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 B 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 {*log*} doesn't evaluate arithmetic expressions 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.

4.2 Translating ordered pairs

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

If in B we have $p \in X \times Y$ then the {*log*} type declaration for p is `dec(P,[x,y])`, where `x` corresponds to the encoding of type X in {*log*}; similarly for `y`.

4.3 Translating sets

4.3.1 Extensional sets — Introduction to set unification

In {*log*} the empty set is written as in B, `{}`. 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 B and {*log*}. For example, if in B x is a variable, then the set $\{2,x,6\}$ is translated as `{2,X,6}`; and if in B *Run* is an element of a set declared in the SETS section, then the set $\{2,Run,6\}$ is translated as `{2,run,6}`.

However, {*log*} provides a form of extensional sets that, in a sense, is more powerful than the one offered in B. 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 B assignment (assume d is a variable):

$$A := A \setminus \{d\}$$

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

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

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

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

which in general is more efficient.

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

Why we conjoined `D nin A_`? Simply because, for instance, `A = {1,2}`, `D = 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 nin A_` is conjoined all the solutions where `D` 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.

4.3.2 Cartesian products

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

4.3.3 Integer intervals

A B integer interval such as $m..n$ is translated as `int(m,n)`. `m` and `n` can be integer constants or variables. If we need to write something like $m+1..2*n+3$ we do as follows: `int(K,J) & K is M + 1 & J is 2*N + 3`, where `K` and `J` must be new variables.

4.4 Translating set and relational 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.

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`.

4.5 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 B it's necessary to add predicates of the form

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)`	$\mathrm{dom}\, R = A$
range	`ran(R,A)`	$\mathrm{ran}\, R = A$
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)`	$S = A \lhd\!\!\!\!- R$
range restriction	`rres(A,R,S)`	$S = R \rhd A$
range anti-restriction	`rares(A,R,S)`	$S = R \rhd\!\!\!\!- A$
update	`oplus(R,S,T)`	$T = R \oplus S$
relational image	`rimg(R,A,B)`	$B = R[A]$
Negations		
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]}`	$\langle a,b,\ldots,z\rangle$
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)`	$t = s \upharpoonright A$

Table 3: Sequence operators available in {*log*}

$x \in \mathrm{dom} f$, before attempting to apply f to x. The translation of these formulas into {*log*} can be done by using the predicate `applyTo` or by using a set membership predicate which leads to set unification. For example the B formula:

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

can be translated in a direct fashion:

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

or just using `applyTo`:

```
applyTo(F,X,Y)
```

or using set unification (if we *assume* that `F` is a function):

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

The {*log*} definition of `applyTo` is the following:

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

If we know that $x \in \mathrm{dom} f$ the there exist `Y` and `G` such that `F = {[X,Y] / G} & [X,Y] nin G`. 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 on 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): `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`.

Therefore, `applyTo(F,X,Y)` implies that `X` belongs to the domain of `F`. If this is not the case then `applyTo(F,X,Y)` fails. Then, if we have to translate $x \in \mathrm{dom} f \wedge f(x) = y$ it's enough to state `applyTo(F,X,Y)`.

However, if in B we have that $f \in T \nrightarrow U$ is part of the invariant, then $x \in \mathrm{dom}(f) \wedge f(x) = y$ will be defined due to the invariant. That is, $f(x)$ will be a unique value. This means that encoding it as `applyTo(F,X,Y)` is too much because `applyTo` asserts that `F` is locally a function on `X`. Hence, in this case, a more precise encoding is the one based on set unification:

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

Note that this encoding implies that `X` belongs to the domain of `F` (otherwise it will fail as `applyTo`). More importantly, this encoding is saying that all we have to do to find the image of `X` under `F` is to walk through `F` looking for *the* ordered pair whose first component is `X`. On the other hand, the encoding based on `applyTo` is saying that once we have found `[X,Y]` in `F` we have to keep walking through it to check that there's no other pair whose first component is `X`. This last check required by `applyTo` is redundant if we know that `F` is a function. If we have proved that $f \in T \nrightarrow U$ is an invariant then we know for sure that f is a function.

Observe that in the translation of **findBirthday** we have used `applyTo` which, after the above analysis, is not the best choice because *pfun*(*birthday*) is intended to be an invariant of the specification. We should replace `applyTo` by the encoding based on set unification. We didn't do it in that way because we think that it requires a rather complex explanation when we were just introducing {*log*}.

4.6 Translating logical operators

Logical conjunction (`&`), disjunction (`or`), implication (`implies`) and negation (`neg`) are among the available logical connectives in {*log*} (see Section 3.3 of the {*log*} manual[1] for the complete list). Logical negation (`neg`) must be used with care because, as the manual explains in Section 3.3, it doesn't work well in all cases. In general, `neg` works as expected when the formula to be negated doesn't contain existential variables inside it. For instance, the following formula states that `Min` is the minimum element in `S`:

```
Min in S & subset(S,int(Min,Max))
```

`neg` won't work correctly for this formula because `Max` is an existential variable inside the formula. In order to see that `Max` is an existential variable inside the formula, we can write it as the body of a clause computing the minimum element of a set:

```
min(S,Min) :- Min in S & subset(S,int(Min,Max)).
```

Now it's clear that `Max` is an existential variable inside the formula because it's not an argument of the clause head. Hence, `neg` won't work well for `min`. More precisely, if we define the clause `n_min` as follows:

```
n_min(S,Min) :- neg(Min in S & subset(S,int(Min,Max))).
```

it doesn't correspond to $\neg$ `min(S,Min)` because `neg` won't compute the (correct) negation of its argument as it contains `Max`. `neg` will compute some formula but not the negation we're expecting.

[1] https://www.clpset.unipr.it/SETLOG/manual_4_9_8.pdf

On the other hand, {*log*} provides the negation for all its atomic constraints (Tables 1-2 and all the arithmetic constraints). `neg` works correctly for all of them. For example, if we want to translate $\neg\, x \in A$ we can write in {*log*} `neg(X in A)` or just `X nin A`. In the same way, $\neg\, A = b$ can be translated as `neg(A neq b)` or as `A neq b`. For instance, the B predicate $A \not\subseteq B$ is translated as `nsubset(A,B)`; and $\neg\, a \leq y$ as `neg(A =< Y)`. Tables 1-2 include the negation for every set theoretic operator.

As an example of using `neg`, the following B statement:

$$\textbf{IF } x \in \mathrm{dom}(f) \wedge 0 < x \textbf{ THEN } f, msg := \{x\} \lhd f, ok \textbf{ ELSE } msg := error \textbf{ END}$$

can be translated as follows:

```
dom(F,D) &
(X in D & 0 < X & dares({X},F,F_) & Msg = ok
or
 neg(X in D & 0 < X) & Sa_ = Sa & Msg = error
)
```

Note that `dom(F,D)` is placed outside the disjunction because the constraint is used to name the domain of `F`. Observe that `D` isn't present in the B statement; it has to be introduced in {*log*} to name the expression dom(f). `dom(F,D)` states that `D` is the (name of the) domain of `Sa`: it makes no sense to negate this because we're *defining* `D` as such. This situation arises frequently when a B specification is translated into {*log*} due to the fact that B uses expressions for what in {*log*} is written with predicates.

4.6.1 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 B we have:

$$\exists x.(x \in \mathbb{N} \wedge x \in A)$$

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:

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

whose semantics is:

$$\forall x.(x \in A \Rightarrow P(x))$$

which, as can be seen, coincides with the universally quantified predicates available en B.

In {*log*} the simplest RUQ are encoded as follows:

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

There are more complex and expressive RUQ available in {*log*}[2].

Recall that a proper use of the B language tends to avoid most of the quantified formulas.

4.7 Translating $\mathbb{N}$

As $\mathbb{N}$ is not a type and, at the same time, is an interpreted set, we must be careful when translating $\mathbb{N}$ into {*log*}.

A type declaration such as $x \in \mathbb{N}$ is equivalent to $x \in \mathbb{Z} \land 0 \leq x$. As we have said, $x \in \mathbb{Z}$ is encoded in terms of the type system defined in {*log*}, whereas $0 \leq x$ is simply encoded as `0 =< X`. On the other hand, $A \subseteq \mathbb{N}$ or $A \in \mathbb{P}\mathbb{N}$ are translated with a RUQ:

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

In particular a type declaration such as $f \in T \nrightarrow \mathbb{N}$ is encoded in {*log*} as follows:

```
pfun(F) & foreach([X,Y] in F, 0 =< Y)
```

plus a type declaration for f such as `dec(F,rel(t,int))`, assuming T is a basic type.

5 Running {*log*} forgrams

{*log*} forgrams usually won't meet the typical performance requirements demanded by users. Forgrams are slower than programs but they have computational properties that programs don't. Hence, we see a {*log*} forgram of a B specification more as a *prototype* than as a final program. On the other hand, given the similarities between a B specification and the corresponding {*log*} forgram, 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 caused 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 put in the market, it implies to correct the error in the requirements document, the specification, the design, the implementation, the user documentation, etc.

Since we see {*log*} forgrams 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 executing a piece of code. The word *simulation* is usually used in the context of *models* (e.g. modeling and simulation). In a sense, our {*log*} forgrams 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 B specification can be seen

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

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

as an *executable specification*. In fact, as we will see, {*log*} forgrams 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 forgram, model or specification and observe the produced outputs or effects. Besides, we will show that {*log*} offers more possibilities beyond this basic idea.

5.1 Basic simulations

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

```
~/setlog$ prolog

?- consult('setlog.pl').

?- setlog.

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

{log}=> birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_,M).
K = {},
B = {},
K_ = {maxi},
B_ = {[maxi,160367]},
M = ok

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

The meaning of the above code is 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_,M).`

   consisting of:
   - `birthdayBookInit` is called passing to it any two variables as arguments;

[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.

- `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 three new variables in the last three arguments.

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_,M).
```

When we call `birthdayBookInit(K,B)`, `K` and `B` unify with `Known` and `Birthday` which are the formal arguments 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_,M)` is called, it's like we were calling:

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

Calling `addBirthday` makes {*log*} to execute each branch of the disjunction present in the body of the clause. That is, both branches are tried in the order they're written. Then, unification goes as follows:

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

Hence the code in the first branch is instantiated as follows:

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

which reduces to:

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

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

When 'y' is pressed {*log*} executes the second branch. Again, unification takes place and a new series of equations are produced:

```
Known = {}
Birthday = {}
Name = maxi
K_ = Known
B_ = Birthday
M = Msg
```

which implies that K unifies with {}. Then, the code in the second branch is instantiated as follows:

```
maxi in {} ...
```

As this predicate is obviously false, the invocation of this branch 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,M1) &
addBirthday(K1,B1,'Yo',201166,K2,B2,M2)    & findBirthday(K2,B2,'Yo',C,K3,B3) &
addBirthday(K3,B3,'Otro',201166,K4,B4,M4) & 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]},
M1 = ok,
K2 = {maxi,Yo},
B2 = {[maxi,160367],[Yo,201166]},
M2 = ok,
C = 201166,
K3 = {maxi,Yo},
B3 = {[maxi,160367],[Yo,201166]},
K4 = {maxi,Yo,Otro},
B4 = {[maxi,160367],[Yo,201166],[Otro,201166]},
M4 = ok,
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 forgram 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,M).
```

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 towards the end of the call. 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,M1) & 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]},
M1 = ok,
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 5.6.

5.1.1 Hiding the complete trace of the execution

If we don't need the complete execution trace of a simulation but only the its final state and outputs we can define a clause for the simulation whose arguments are the variables we are interested in:

```
sim(K_,B_,C,Card,Card1) :-
  birthdayBookInit(K,B) &
  addBirthday(K,B,maxi,160367,K1,B1,M1) &
  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_).
```

And then we call the clause:

```
{log}=> sim(K_,B_,C,Card,Card1).
K_ = {maxi,Yo,Otro},
```

```
B_ = {[maxi,160367],[Yo,201166],[Otro,201166]},
C = 201166,
Card = {maxi},
Card1 = {Yo,Otro}
```

As can be seen, we get a more compact output showing only the variables we are interested in.

5.2 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 on {*log*}'s types.

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.pl').

?- setlog.

{log}=> type_check.         % typechecker is active

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

In this way, when {*log*} executes `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 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_,M).

***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:160367,K_,B_,M) &
        dec([K,K_],kn) & dec([B,B_],bb) & dec(M,msg).

K = {},
B = {},
K_ = {name:maxi},
B_ = {[name:maxi,date:160367]},
M = ok
```

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 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`).

5.3 Simulations 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_solver(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 `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:

```
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:

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

```
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).

5.4 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 execution 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 execute forgrams, within certain limits. These limits are given by set theory and non-recursive clauses. The following are the conditions under which {*log*} can perform symbolic executions[8]:

1. Recursive clauses are not allowed.
2. Only the operators of Tables 1 and 2 are allowed. If the {*log*} forgrams 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 executions are to be done[10]. Actually, many symbolic executions are still possible even if the above conditions aren't met.

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

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

[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 executions. These conditions are more or less complex and quite technical. The {*log*} forgrams that can't be symbolically simulated and don't verify the following conditions will not appear in this course.

[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 fully implemented in {*log*}, yet.

in which case {*log*} answers:

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

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,N1,C1,K1,B1,M1) & addBirthday(K1,B1,N2,C2,K_,B_,M2).
```

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

```
K = {},
B = {},
K1 = {N1},
B1 = {[N1,C1]},
M1 = ok,
K_ = {N1,N2},
B_ = {[N1,C1],[N2,C2]},
M2 = ok
Constraint: N1 neq N2
```

As can be seen, the answer includes the `Constraint` section 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 `[N2,C2]` to the birthday book if and only if `N2 nin {N1}`, which holds if and only if `N2` is different from `N1`.

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

```
K = {},
B = {},
K1 = {N1},
B1 = {[N1,C1]},
M1 = ok,
N2 = N1,
K_ = {N1},
B_ = {[N1,C1]},
M2 = nameExists
```

produced after considering that `N1` and `N2` are equal in which case the second invocation to `addBirthday` goes through the **ELSE** branch 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 this:

```
birthdayBookInit(K,B) & addBirthday(K,B,N1,C1,K1,B1,M1) &
addBirthday(K1,B1,N2,C2,K2,B2,M2) & findBirthday(K2,B2,W,X,K2,B2).
```

{*log*} will consider several particular cases depending on whether `N2`, `N1` and `W` are equal or not. For example, the following are the first three solutions returned by {*log*}:

```
K = {},
B = {},
K1 = {N1},
B1 = {[N1,C1]},
M1 = ok,
K2 = {N1,N2},
B2 = {[N1,C1],[N2,C2]},
M2 = ok,
W = N1,
X = C1
Constraint: N1 neq N2

Another solution?  (y/n)
K = {},
B = {},
K1 = {N1},
B1 = {[N1,C1]},
M1 = ok,
K2 = {N1,N2},
B2 = {[N1,C1],[N2,C2]},
M2 = ok,
W = N1,
X = C1
Constraint: C1 neq C2, N1 neq N2

Another solution?  (y/n)
K = {},
B = {},
K1 = {N1},
B1 = {[N1,C1]},
M1 = ok,
K2 = {N1,N2},
B2 = {[N1,C1],[N2,C2]},
M2 = ok,
W = N2,
X = C2
Constraint: N1 neq N2
```

In the first case `W = N1` is considered and so `X` must be equal to `C1`; the second case is similar to the first one; and in the third `W = N2` and so `X` is equal to `C2`. {*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.

5.5 Inverse simulations

Normally, in a simulation the user provides inputs and the forgram 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 executions within the same limits in which it is able to perform symbolic executions. 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 execute a forgram by providing values for the input variables or for the output variables; in fact, {*log*} is able to execute a forgram 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_,M).
```

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,
M = ok
```

When the B specification is deterministic, the corresponding {*log*} forgram will be deterministic as well. Therefore, for any given input there will be only one solution. However, the inverse simulation of a deterministic forgram may generate a number of solutions. This is the case with the above simulation. The first solution computed by {*log*} considers the case where `N = alvaro` and `C = 110400`, but this isn't the only possibility. Going forwards with the solutions 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,
M = ok
```

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.

5.6 Evaluation of predicates

At the end of Section 5.1 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(Known,Birthday)`.

Hence, we can check whether or not the above state satisfies the invariant by asking {*log*} to solve the following:

```
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`, as in the following example (note that `maxi` is missing from `known`):

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

6 Proving the correctness of {*log*} forgrams

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 programming language. However, {*log*} is also a *satisfiability solver*[11]. 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 given by the operators listed in Tables 1 and 2, and combined with linear integer arithmetic[12].

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

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

[11] See for instance Wikipedia: Satisfiability modulo theories.

[12] 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.

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 is unsatisfiable it means that there is no finite set satisfying it. Formally, F is an unsatisfiable formula if:

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

where y ranges over all finite sets. 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.

If {*log*} is called on some formula there are four possible behaviors:

1. {*log*} returns `no`. This means the formula is unsatisfiable.
2. {*log*} returns one or more solutions. This means the formula is satisfiable. For example, the simulations we run in Section 5 are all satisfiable formulas.
3. {*log*} returns a warning messages. This means the answer is unreliable. We can't be sure whether the formula is satisfiable or not.
4. {*log*} doesn't seem to return. You wait in front of the screen after pressing the return key but no answer is produced; you wait longer but still nothing happens. This means that {*log*} is unable to determine whether the formula is satisfiable or not. This in turn may occur because the formula is too complex and makes {*log*} to take a very long time of just because {*log*} enters into an infinite loop. Situations like this are rare and usually occur in complex problems. If you want to see this behavior try the following:

   ```
   comp(R,R,R) & [X,Y] in R & [Y,Z] in R & [X,Z] nin R.
   ```

   What is the meaning of this formula?

One important aspect is that {*log*}, as other satisfiability solvers, *automatically* decides the satisfiability of a given formula. That is, no action from the user is required. Hence, when {*log*} finds that F is unsatisfiable it has *automatically proved* the theorem $\neg\, F$. This is called *automated theorem proving* which is part of *automated software verification*. There are, however, automated theorem provers that aren't satisfiability solvers[13]. Satisfiability solvers and automated theorem provers can be used to prove mathematical theorems but we're interested in their application to software verification.

More specifically, we're going to apply {*log*}'s capabilities for automated theorem proving to ensure machine consistency. Recall that in Section 5 of "Introduction to the B-Method" we

[13] See for instance Wikipedia: Automated theorem proving.

show that the B-Method requires to discharge some proof obligations once we have written a B machine. Then, we're going to use {*log*} to discharge those proof obligations on the corresponding {*log*} forgram. That is, once we have translated the B specification into {*log*}, we're going to use {*log*} to generate the same proof obligations required by the B-Method and then we're going to use {*log*} again to *automatically* discharge them. This process implies that the forgram so verified becomes a *certified prototype* of the system. In other words, the forgram is an implementation verifying all the the verification conditions set forth by the B-Method.

6.1 Invariance lemmas in {*log*}

The most complex verification conditions required by the B-Methods are the invariance lemmas. Recall that an invariance lemma states that each operation of a B specification preserves the state invariant. Formally, if an operation depends on an input parameter x, has precondition *Pre* and changes state variable v with *Post*, the invariance lemma is as follows:

$$\forall x.(Inv \wedge Pre \Rightarrow Inv[v \mapsto Post])$$

In turn, when this operation is translated as a {*log*} clause we have $v\_$ as the next-state variable. The abstract assignment $v := Post$ becomes an equality of the form $v\_ = Post$. Therefore, the invariance lemma can be written as follows:

$$\forall x.(Inv \wedge Pre \Rightarrow Inv[v \mapsto v\_])$$

If we define $Inv\_$ as a shorthand for $Inv[v \mapsto v\_]$, then we have:

$$\forall x.(Inv \wedge Pre \Rightarrow Inv\_)$$

Recall that in order to prove the above formula in {*log*} we must negate it:

$$\neg\ (\forall x.(Inv \wedge Pre \Rightarrow Inv\_))$$

At the same time during the translation of the B-Machine into {*log*}, we have split the invariance in several pieces. Recall that for the birthday book specification we have the following:

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

Then, for instance, this is the invariance lemma for `addBirthday`:

```
addBirthday_pi_birthdayBookInv :-
  neg(
    birthdayBookInv(Known,Birthday) &
    addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) implies
    birthdayBookInv(Known_,Birthday_)
  ).
```

The idea is that the user executes `addBirthday_pi_birthdayBookInv` and {*log*} answers `no`. As we have said above, this means that {*log*} couldn't find values for the variables as to satisfy the formula (i.e. the formula is unsatisfiable). In turn, as we have explained, this means that the formula inside `neg` is a theorem and so {*log*} has discharged this proof obligation.

There's, though, a problem that we need to address. Internally, {*log*} transforms the body of `addBirthday_pi_birthdayBookInv` in:

```
birthdayBookInv(Known,Birthday) &
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) &
neg( birthdayBookInv(Known_,Birthday_) ).
```

because $\neg\,(I \wedge T \Longrightarrow I\_) \equiv \neg\,(\neg\,(I \wedge T) \vee I\_) \equiv I \wedge T \wedge \neg\, I\_$. The problem is that {*log*} can't compute the negation of user-defined clauses. Then, {*log*} will issue a warning such as:

```
***WARNING***: Unsafe use of negation - using naf
```

In order to avoid this problem we have to help {*log*} to compute the negation of the clauses declared as invariants. More precisely, we have to add the following to the birthday book forgram:

```
dec_p_type(n_birthdayBookInv(kn,bb)).
n_birthdayBookInv(Known,Birthday) :- neg(dom(Birthday,Known) & pfun(Birthday)).
```

That is, for each clause `p` declared as an invariant, a clause named `n_p` with the same arity and whose body is the negation of `p`'s body, is added to the forgram. In this way when {*log*} has to compute `neg(birthdayBookInv(Known_,Birthday_))` it looks up among the clauses one whose head is `n_birthdayBookInv` and with `birthdayBookInv`'s arity. If such a clause is present, {*log*} uses its body to compute the negation; otherwise it issues a warning message such as the one above. These clauses are called *negative clauses*. Note that negative clauses aren't declared as invariants although their types are those of the corresponding positive clauses. See Appendix A for the complete forgram implementing the birthday book.

Recall that `neg` doesn't always work correctly, as we explained in Section 4.6. However, it works well in many cases. You won't see problems with `neg` in what concerns the exercises of this course. You can have a look at the problem of computing $\neg\, p$ in logic programming in Wikipedia: Negation as failure.

In any case, if you are in front of a formula for which `neg` doesn't work well, you can manually write its negation and put it in a negative clause. To that end you have to distribute the negation all the way down to the atoms at which point you use the negations of the operators of Tables 1 and 2.

6.2 The verification condition generator (VCG)

{*log*} can automatically generate verification conditions similar to those required by the B-Method, plus some more not required by the B-Method. That is, {*log*} generates verification conditions as those discussed in Section 5 of "Introduction to the B-Method". We'll exemplify the process to generate verification conditions with the birthday book forgram.

```
~/setlog$ prolog

?- consult('setlog.pl').

?- setlog.

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

VGC stands for *verification condition generator*. The command takes as argument the name of a file containing a forgram implementing a state machine (in particular one resulting from the translation of a B machine). That is, the forgram must have declarations such as `variables`, `invariant`, etc. as described in Section 2 and in chapter 11 of the {*log*} user's manual. VCG checks some well-formedness conditions on the forgram as described in detail in the referred manual. If all these checks are passed then VCG generates a file named, for instance, `bb-vcg.slog`. Appendix B lists the contents of `bb-vcg.slog` as produced by VCG.

Once VCG has been called on a file, the user has to consult the file generated by VCG and run the command indicated by {*log*}:

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

Type checking has been deactivated.


Call check_vcs_bb to run the verification conditions.

file bb-vc.pl consulted.
```

As can be seen, {*log*} says that we should call `check_vcs_bb` to run or discharge the verification conditions. This command is always of the form `check_vcs_<filename>`. If we run the command we'll see the following:

```
{log}=> check_vcs_bb.

Checking birthdayBookInit_sat_birthdayBookInv ... OK
Checking addBirthday_is_sat ... OK
Checking findBirthday_is_sat ... OK
Checking remind_is_sat ... OK
Checking addBirthday_pi_birthdayBookInv ... OK
Checking findBirthday_pi_birthdayBookInv ... OK
Checking remind_pi_birthdayBookInv ... OK
```

As you can see, {*log*} is able to automatically discharge all proof obligations. However, this might not always be the case. Why {*log*} might be unable to discharge a proof obligation and how to remedy this situation is explained in the next section.

VCG generates basically two classes of verification conditions:

- Satisfiability Conditions. These are identified by the word `_sat_`. For example, `addBirthday_is_sat` and `birthdayBookInit_sat_birthdayBookInv`.

  The expected answer for a satisfiability condition is a solution. In other words, if {*log*} answers `no` for such a verification condition there's an error in the specification.

- Invariance Lemmas. These are identified by the word `_pi_` (for "preserves invariant"). For example, `addBirthday_pi_birthdayBookInv`.

  The expected answer for an invariance lemma is `no`. In other words, if {*log*} returns a solution for such a verification condition there's an error in the specification.

6.3 When {*log*} fails to discharge a proof obligation

We'll focus this section on invariance lemmas but similar conclusions can be drawn for satisfiability conditions. {*log*} may fail to discharge (i.e. prove) an invariance lemma, basically, for two reasons:

1. The invariant is wrong. In this case, the invariant is either too strong or too weak. If it's too strong, it means that you're asking too much to your system. You want your system to verify some invariant but it can't. For example, the following is too strong for the savings account system:

   $$sa \in NIC \leftrightarrow \mathbb{Z} \wedge pfun(sa) \wedge \forall x, y.(x \mapsto y \in sa \Rightarrow 0 < y)$$

   If it's too weak it means that you're allowing some operations to be called from states they don't expect to be called. For example, the following is too weak for the birthday book:

   $$birthday \in NAME \nrightarrow DATE$$

2. The operation is wrong. The most common situation is to have a weaker precondition than needed. For example, the following specification of **addBirthday** has a precondition making the operation to fail to verify $birthday \in NAME \nrightarrow DATE$:

   $msg \leftarrow$ **addBirthday**$(name, date) \mathrel{\widehat{=}}$
   PRE $name \in NAME \wedge date \in DATE$
   THEN $known, birthday, msg := known \cup \{name\}, birthday \cup \{name \mapsto date\}, ok$
   END

   Can you tell why? Can you provide a counterexample?

In order to see how {*log*} behaves when it fails to prove an invariance lemma, let's assume that the invariant for the birthday book is just: `pfun(Birthday)`. In this case the invariance lemma for `addBirthday` is as follows:

```
neg(
  pfun(B) &
  addBirthday(K,B,N,C,K_,B_,M) implies
  pfun(B_)
).
```

When {*log*} is asked to solve the above formula the answer is the following:

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

As the above formula is satisfiable (which means that the formula inside `neg` isn't a theorem), {*log*} returns a solution that, in this case, is read as a *counterexample*. That is, {*log*} returns an assignment of values to variables showing that `addBirthday` doesn't preserve the invariant.

By analyzing the counterexample we can discover why `addBirthday` fails to preserve the invariant giving us the chance to fix the error. The first thing we can do to analyze the counterexample is to replace all the set variables by the empty set[14]. After a little bit of

[14]Except those at the left-hand side of the equalities.

simplification we obtain:

```
B = {[N,_N2]},
K = {},
K_ = {N},
B_ = {[N,C],[N,_N2]},
M = ok
Constraint: C neq _N2
```

Observe that {*log*} considers executing `addBirthday` with `B = {[N,_N2]}` and `K = {}`. This clearly violates dom(*birthday*) = *known*. Actually, if we add this condition to the invariance lemma, {*log*} returns `no`.

```
neg(
  pfun(B) & dom(B,K) &                        %%% new condition
  addBirthday(K,B,N,C,K_,B_,M) implies
  pfun(B_)
).
```

Clearly, now {*log*} can't execute `addBirthday` from a state not verifying dom(*birthday*) = *known*.

Recall that in Section 2.3.3 we said that the B invariant can be encoded in {*log*} as several clauses (one for each conjunct in the INVARIANT section). In this case, {*log*} may fail to prove some invariance lemmas because it needs some of the other invariants as hypothesis. Think that if we separate the invariant of the birthday book in two clauses as we suggest at the end of Section 2.3.3, {*log*} won't be able to prove that `addBirthday` preserves `pfun(Birthday)` for the same reason analyzed above. The missing hypothesis can be manually conjoined to the invariance lemmas generated by VCG.

FORGRAMS

What is a *forgram*? Forgram is a portmanteau word resulting from the combination of *for*mula and pro*gram*. A forgram is a piece of code that enjoys the *formula-program duality*. In other words, a forgram is a piece of code that can be used as a formula *and* as a program. In Section 5 we showed that {*log*} code can be executed as a program; and in Section 6 we showed that {*log*} code can be used as a formula. In {*log*} engineers write forgrams, instead of plain programs.

References

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

A The {*log*} forgram of the birthday book

```
variables([Known,Birthday]).

def_type(bb,rel(name,date)).
def_type(kn,set(name)).
def_type(msg,enum([ok,nameExists])).

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

dec_p_type(n_birthdayBookInv(kn,bb)).
n_birthdayBookInv(Known,Birthday) :- neg(dom(Birthday,Known) & pfun(Birthday)).

initial(birthdayBookInit).
dec_p_type(birthdayBookInit(kn,bb)).
birthdayBookInit(Known,Birthday) :- Known = {} & Birthday = {}.

operation(addBirthday).
dec_p_type(addBirthday(kn,bb,name,date,kn,bb,msg)).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) :-
  (Name nin Known &
   un(Known,{Name},Known_) &
   un(Birthday,{[Name,Date]},Birthday_) &
   Msg = ok
  or
   Name in Known &
   Known_ = Known &
   Birthday_ = Birthday &
   Msg = nameExists
  ).

operation(findBirthday).
dec_p_type(findBirthday(kn,bb,name,date,kn,bb)).
findBirthday(Known,Birthday,Name,Date,Known,Birthday) :-
  Name in Known & applyTo(Birthday,Name,Date).

operation(remind).
dec_p_type(remind(kn,bb,date,kn,kn,bb)).
remind(Known,Birthday,Today,Cards,Known,Birthday) :-
  rres(Birthday,{Today},M) & dom(M,Cards) & dec(M,bb).
```

B File generated by VCG for the birthday book

```
% Verification conditions for bb.slog

% Run check_vcs_bb to see if the program verifies all the VCs

:- notype_check.

:- consult('bb.slog').

birthdayBookInit_sat_birthdayBookInv :-
  birthdayBookInit(Known,Birthday) &
  birthdayBookInv(Known,Birthday).

addBirthday_is_sat :-
  addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) &
  [Known,Birthday] neq [Known_,Birthday_].

addBirthday_pi_birthdayBookInv :-
  neg(
    % here conjoin other invariants as hypothesis if necessary
    birthdayBookInv(Known,Birthday) &
    addBirthday(Known,Birthday,Name,Date,Known_,Birthday_,Msg) implies
    birthdayBookInv(Known_,Birthday_)
  ).

findBirthday_is_sat :-
  findBirthday(Known,Birthday,Name,Date,Known,Birthday).

findBirthday_pi_birthdayBookInv :-
  % findBirthday doesn't change birthdayBookInv variables
  neg(true).

remind_is_sat :-
  remind(Known,Birthday,Today,Cards,Known,Birthday).

remind_pi_birthdayBookInv :-
  % remind doesn't change birthdayBookInv variables
  neg(true).

check_sat_vc(VCID) :-
  write('\nChecking ') & write(VCID) & write(' ... ') &
  ((call(VCID) & write_ok)!
   or
   write_err
```

```
  ).

check_unsat_vc(VCID) :-
  write('\nChecking ') & write(VCID) & write(' ... ') &
  ((call(naf(VCID)) & write_ok)!
   or
   write_err
  ).

write_ok :-
  prolog_call(ansi_format([bold,fg(green)],'OK',[])).

write_err :-
  prolog_call(ansi_format([bold,fg(red)],'ERROR',[])).

check_vcs_bb :-
  check_sat_vc(birthdayBookInit_sat_birthdayBookInv) &
  check_sat_vc(addBirthday_is_sat) &
  check_sat_vc(findBirthday_is_sat) &
  check_sat_vc(remind_is_sat) &
  check_unsat_vc(addBirthday_pi_birthdayBookInv) &
  check_unsat_vc(findBirthday_pi_birthdayBookInv) &
  check_unsat_vc(remind_pi_birthdayBookInv) &
  true.

:- nl &
   prolog_call(ansi_format([bold,fg(green)],
               'Type checking has been deactivated.',[])) &
   nl & nl.

:- nl &
   prolog_call(ansi_format([bold,fg(green)],
               'Call check_vcs_bb_b_inv to run the verification conditions.',
               [])) &
   nl & nl.
```