Rapid Prototyping and Animation of Z Specifications Using {*log*}

Maximiliano Cristiá[1] and Gianfranco Rossi[2]

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

Abstract. Prolog has been proposed as the programming language on which animation and prototyping of Z specifications should be based. However, we believe there is still room for improvements. In this paper, we want to revisit this issue in the light of a powerful, set-oriented constraint programming language like {*log*} (pronounced 'setlog'). In particular, we pay attention to three points that we think are crucial: finding solutions to complex state predicates; defining formal criteria for guiding an evaluation process based on prototypes; and automatically adding graphical user interfaces to prototypes generated from Z specifications. Three examples of information systems prototypes are available online.

1 Running Example

Assume you are eliciting the user requirements for an ERP system. In a first meeting, your customer asks the following:

(R1) Keep a record of client accounts where the company registers all the credits and debits of its clients.
(R2) Get a listing of all the accounts whose balance is in a given range.

Given that you have experience in requirement engineering, you know that probably your customer has told you (R1) and (R2) without giving them too much thought. You would like to be sure that these are the real, final requirements before engaging your team in programming unstable, unvalidated requirements. Following the software engineering literature you know that it would be of great help if you could show your customer a prototype of these requirements.

Now, suppose you know of a tool that automatically generates *functional* prototypes of some kind of Z specifications, provided they are annotated with user interface directives. Further, let us say that in your team there is an engineer who is knowledgeable in the Z notation. Would it be cost-effective to write a Z specification of (R1) and (R2) and use that tool to generate a prototype so your customer can validate them? In this way, your programmers will implement validated requirements; moreover, they will do it from a formal specification. All this would amount to a much better product with less iterations to improve its

quality. Even more, assume that this prototyping tool records all the interactions of users. Then, later, these interactions can be used as *functional* test cases. So the tool would also help you during the testing process.

After considering all these gains, you decide to ask your team to write the Z specification shown in Figure 1 and the GUI specification shown in 3 which, when fed into the prototyping tool, become the Prolog+{*log*}+XPCE program [25, 11, 36] partially shown in Figure 4. When this program is executed the user can play with an application presenting an acceptable GUI like the one shown in Figure 2—download this and two more examples from [6].

$$[UID, NAME]$$

$$ERP == [clients : UID \pfun NAME;\ accounts : UID \pfun \mathbb{Z} \mid \mathrm{dom}\ clients = \mathrm{dom}\ accounts]$$

$$InitERP == [ERP \mid clients = \emptyset \land accounts = \emptyset]$$

NewClientOk

$$\Delta ERP;\ u? : UID;\ n? : NAME$$

$$u? \notin \mathrm{dom}\ clients$$
$$clients' = clients \cup \{u? \mapsto n?\}$$
$$accounts' = accounts \cup \{u? \mapsto 0\}$$

TransactionOk

$$\Delta ERP;\ n? : UID;\ m? : \mathbb{Z}$$

$$n? \in \mathrm{dom}\ accounts \land m? \neq 0$$
$$accounts' = accounts \oplus \{n? \mapsto accounts\ n? + m?\}$$
$$clients' = clients$$

$$ClientAlreadyExists == [\Xi ERP;\ u? : UID \mid u? \in \mathrm{dom}\ clients]$$
$$NewClient == NewClientOk \lor ClientAlreadyExists$$
$$AccountNotExists == [\Xi ERP;\ n? : UID \mid n? \notin \mathrm{dom}\ accounts]$$
$$NullAmount == [\Xi ERP;\ m? : \mathbb{Z} \mid m? = 0]$$
$$Transaction == TransactionOk \lor AccountNotExists \lor NullAmount$$
$$AccountsInRangeOk == [\Xi ERP;\ a?, b? : \mathbb{Z};\ r! : \mathbb{P}\ UID \mid a? \leq b? \land r! = \mathrm{dom}(accounts \rhd (a? \mathinner{..} b?))]$$
$$NotARange == [\Xi ERP;\ a?, b? : \mathbb{Z};\ r! : RET \mid a? > b?]$$
$$AccountsInRange == AccountsInRangeOk \lor NotARange$$

Fig. 1: Excerpt of the Z specification of (R1) and (R2)

Z and {*log*} will be introduced in sections 2 and 3, respectively, and how prototypes would be automatically generated in Sect. 4. Now we want to show

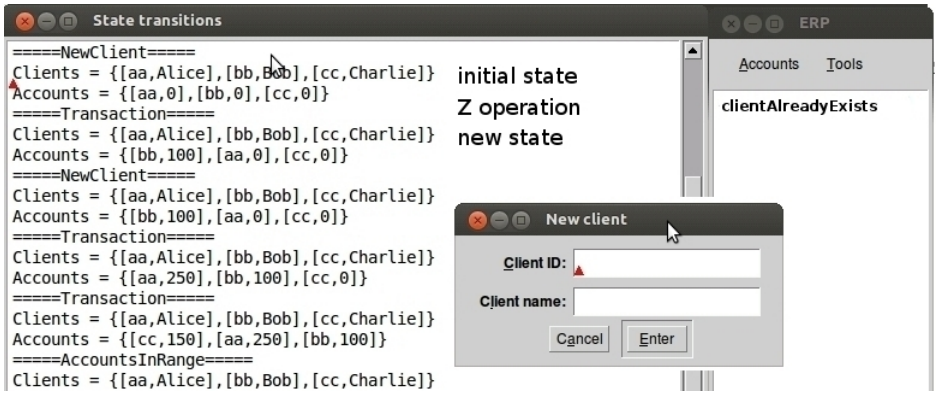

Fig. 2: Screenshot of the prototype

what users and software engineers can do with these prototypes. Note that the GUI is divided into two panels. The one on the left, STATE TRANSITIONS, displays all the state transitions executed while the application is used. It displays the initial state by giving the value for each state variable, the name of the Z operation (transition) executed and the resulting new state—input and output variables are not displayed to keep the example manageable but they can be trivially added. The panel on the right, named ERP in this case, is the actual prototype which presents a menu bar. This menu bar contains as many pull-down menus as indicated in the GUI specification plus the TOOLS menu automatically added to all prototypes. Although not shown in Figure 2, the ACCOUNTS menu contains three options (NEW CLIENT, INFORM TRANSACTION and ACCOUNTS IN RANGE) as indicated by the GUI specification given in Figure 3. The TOOLS menu contains only one option called SET STATE. When users click on a given menu option a dialog box opens, as, for instance, the one shown in the figure which corresponds to the NEW CLIENT option. Again, dialog boxes display GUI widgets according to the GUI specification, plus default Cancel and Enter buttons. Once a dialog box is filled in and the Enter button is pressed, the dialog calls the corresponding Z operation which in turn updates the state and prints some output. Output can be printed in the same dialog-box or in the application main window, as can be seen in the figure. In summary, the prototype executes all the specified operations through a reasonably real GUI.

Allowing end-users to play with the prototype as with a final application is important but they should do it trying to see if you (the requirement engineer) have correctly understood what they asked you for. For instance, executing *AccountsInRange* from the initial state of the system (Figure 1) says little about whether requirement (R2) has been correctly understood or not, because the answer will be the empty set regardless of the values of $a?$ and $b?$. Therefore, it would be better if the operation is executed when the state of the system contains some client accounts with different balances. One possible way of doing this is entering some clients and transactions by using the different menus and options.

Clearly, this is annoying, time consuming and error prone. Alternatively, you can think of a state predicate such that every possible solution would enable satisfactory executions of the operation. For example, solutions to $\#\,\mathrm{ran}\,accounts > 3$ are partial functions that contain at least four elements in the range, which implies at least three elements in the domain. Then, if the state from which *AccountsInRange* is executed satisfies that formula, $\mathrm{dom}(accounts \rhd (a? \,..\, b?))$ (see *AccountsInRangeOk*) can be a set with different numbers of elements depending on the values set for $a?$ and $b?$.

The SET STATE option in the TOOLS menu allows you to enter a Z predicate depending on the state variables of the system. $\{log\}$ tries to find a solution for the goal and, if it succeeds, this solution is bound to the state variables. In other words, the system transitions to a state verifying that goal. Most of Z predicates can be translated into $\{log\}$ goals [7]. Hence, if you call SET STATE, enter the $\{log\}$ representation of $\#\,\mathrm{ran}\,accounts > 3$, and call *AccountsInRange* there will be several accounts with different balances making it possible to set interesting values for $a?$ and $b?$, thus generating different listings. This would allow your end-users to validate requirement (R2) through the prototype by evaluating whether its answers are correct with respect to their expectations. Finding solutions to state predicates is further discussed in Sect. 5.

However, why $\#\,\mathrm{ran}\,accounts > 3$ is the proper condition? Are there other conditions that should be sought? How can they be discovered? Model-based testing (MBT) methods face a similar problem. In Sect. 6 we propose to apply the Test Template Framework (TTF) [32], a MBT method tailored to the Z notation, to discover the key conditions to validate user requirements.

Positive side effects. Say that your customer approves the prototype. This implies that all the transitions recorded by the prototype (cf. panel STATE TRANSITIONS) are correct. You can use them to conduct system and acceptance testing because they represent the runs that your customer has validated. If the final system behaves the same with respect to these executions, your customer should be satisfied with the product. Note that these executions can be replayed in the final product because they are actual values for the important variables—however, some semi-automatic *reification* or *refinement* will be needed, see [5] for an approach to this problem. Furthermore, if the TTF is applied to guide the validation process, these runs can be used for unit testing too. Even more: you have a formal specification of the requirements ready to be handed to the programming team. All this should account for a much better product at the price of writing a Z specification.

Warning. The prototype generation tool that we described in this introduction is not fully implemented yet, but we think it is feasible because: translations from Z to Prolog have already been proposed; $\{log\}$ can solve many complex set predicates [7] making it possible to implement the SET STATE feature; and the TTF has been implemented by the Fastest tool which has been applied to several Z specifications to discover complex testing specifications [4]. We believe that the problem is putting all these techniques and tools to work together.

2 Brief Introduction to the Z notation

Z is a formal specification language based on first-order logic and set theory [30]. It can be used to specify software systems in many different ways. However, in this paper we will consider that specifications take the form of state machines, which is how the notation is widely used. This way of writing specifications is based on a set of interpretations and conventions. Then, all the results presented in this paper hold if specifiers follow them. We will introduce some details of Z by describing the meaning of the specification presented in Figure 1, which formalizes part of (R1) and (R2). *UID* and *NAME* are two uninterpreted sorts (called basic or given types in Z). *ERP* is a so-called schema declaring two variables, *clients* and *accounts*. *clients* is a partial function from *UID* into *NAMES*. *ERP* also restricts the possible values of *clients* and *accounts* to those that have equal domains. *clients* represents the set of agents who are currently clients of the company; *accounts* holds the current balance of each client's account. In this way, *ERP* is the state space of the state machine. In Z, ordered pairs are written as $a \mapsto b$, and partial functions are sets of ordered pairs. Then, *InitERP* simply states that in the initial state both partial functions are empty.

NewClientOk is an operation schema because it defines a state transition. This can be noted by the ΔERP expression. ΔERP declares *clients*, *accounts*, $clients'$ and $accounts'$ where $clients'$ and $accounts'$ represent the value of *clients* and *accounts* in the after-state. In Z state transitions can also consume inputs and produce outputs. Inputs variables are decorated with '?' while output ones with '!'. The predicate part of schema *NewClientOk* states the pre-conditions and post-conditions that define the operation. In this case, the agent for whom the account is being opened must be new to the company; this is formalized by asking that his/her ID must not belong to the domain of *clients*. If this is the case, his/her ID and name are added to the client "database"; and a new account whose balance is zero is added. Note how set expressions are used even for partial functions and how the new state is defined. For instance, $accounts' = accounts \cup \{u? \mapsto 0\}$ states that the value of *accounts* in the after-state must be equal to its value in the before-state plus the ordered pair $u? \mapsto 0$.

Schema *ClientAlreadyExists* states that the system must remain the same when the client already exists. This is formalized by the expression ΞERP which implicitly cojoins $clients' = clients \land balances' = balances$ to the schema. The final operation, *NewClient*, is specified as the disjunction of the two previous schemas. *NewClient* is also a schema defined by a schema expression.

Since the specification of the other operations follows a similar structure, we will only comment about the new expressions. $\oplus$ is a relational operator defined as $f \oplus g = (\mathrm{dom}\, g \lhd f) \cup g$ where $\lhd$ is the domain anti-restriction operator. That is, $A \lhd f = \{p : f \mid p.1 \notin A\}$. In turn, $\rhd$, used in *AccountsInRangeOk*, is the domain restriction operator defined as $A \rhd f = \{p : f \mid p.1 \in A\}$. These operators are part of the Z mathematical toolkit (ZMT) [26], which is a rich and expressive collection of mathematical operators supported by Z. In particular, we would like to emphasize that the notion of set is essential in Z because it is used to define binary relations, partial and total functions, sequences and bags.

Therefore, it is tantamount to the success of an animation or rapid prototyping tool for Z specifications to be based on a powerful, efficient set constraint solver.

3 Brief Introduction to {*log*}

{*log*} [9, 10, 25] is a constraint logic programming (CLP) language that extends Prolog with general forms of sets and basic set-theoretic operations in the form of primitive constraints. Sets are primarily designated by *set terms*, that is, terms of one of the forms: {}, whose interpretation is the empty set, or $\{t_1, \ldots, t_n \mid s\}$, where s is a set term, whose interpretation is the set $\{t_1\} \cup \{t_2\} \cup \cdots \cup \{t_n\} \cup s$. The kind of sets that can be constructed in {*log*} are the so-called *hereditarily finite sets*, that is finitely nested sets that are finite at each level of nesting. Note that similarly to Prolog's lists, a set $\{t_1, \ldots, t_n \mid s\}$ can be partially specified, in that either some of its elements $t_1, \ldots, t_n$ or the remaining part s can contain unbound variables (hence "unknowns"). Sets can be also denoted intentionally by set formers of the form $\{\mathsf{X} : \mathsf{exists}([\mathsf{Y_1}, \ldots, \mathsf{Y_n}], \mathsf{G})\}$, where G is a {*log*}-goal (see below) and $\mathsf{X}, \mathsf{Y_1}, \ldots, \mathsf{Y_n}$ are variables occurring in G. Finally, sets can be denoted by *interval terms*, that is terms of the form $\mathsf{int}(\mathsf{a}, \mathsf{b})$, where a and b are integer terms, whose interpretation is the integer interval $[a, b]$.

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

$$1 \ \mathsf{in} \ \mathsf{R} \ \&\ 1 \ \mathsf{nin} \ \mathsf{S} \ \&\ \mathsf{inters}(\mathsf{R}, \mathsf{S}, \mathsf{T}) \ \&\ \mathsf{T} = \{\mathsf{X}\}$$

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

The original collection of set-based primitive constraints has been extended in [22] to include simple integer arithmetic constraints over Finite Domains as provided by CLP(FD) systems (cf. e.g. [24]).

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

Clauses, goals, and programs in {*log*} are defined as usual in CLP. In particular, a {*log*}-*goal* is a formula of the form $\mathsf{B_1} \ \&\ \mathsf{B_2} \ \&\ \cdots \ \&\ \mathsf{B_k}$, where $\mathsf{B_1}, \ldots, \mathsf{B_k}$

are either user-defined atomic predicates, or atomic $\{log\}$-constraints, or disjunctions of either user-defined or predefined predicates, or Restricted Universal Quantifiers (RUQs). Disjunctions have the form $\mathsf{G_1}$ or $\mathsf{G_2}$, where $\mathsf{G_1}$ and $\mathsf{G_2}$ are $\{log\}$-goals, and are dealt with through non-determinism: if $\mathsf{G_1}$ fails then the computation backtracks, and $\mathsf{G_2}$ is considered instead. RUQs are atoms of the form $\mathsf{forall(X\ in\ s, exists([Y_1, \ldots, Y_n], G))}$, where s denotes a set and G is a $\{log\}$-goal containing $\mathsf{X, Y_1, \ldots, Y_n}$. The logical meaning of this atom is $\forall X(X \in s \Rightarrow \exists Y_1, \ldots, Y_n : G)$, that is G represents a property that all elements of s are required to satisfy. When s has a known value, the RUQ can be used to iterate over s, whereas, when s is unbound, the RUQ allows s to be non-deterministically bound to each set satisfying the property G.

The following is an example of a $\{log\}$ program:

$\mathsf{is\_rel(R)}$:- $\mathsf{forall(P\ in\ R, exists([X, Y], P = [X, Y]))}$.
$\mathsf{dom}(\{\}, \{\})$.
$\mathsf{dom}(\{[X, Y]/Rel\}, Dom)$:- $\mathsf{dom(Rel, D)}\ \&\ \mathsf{Dom} = \{\mathsf{X/D}\}\ \&\ \mathsf{X\ nin\ D}$.

This program defines two predicates, is_rel and dom. $\mathsf{is\_rel(R)}$ is true if R is a binary relation, that is a set of pairs of the form $\mathsf{[X, Y]}$. $\mathsf{dom(R, D)}$ is true if D is the domain of the relation R. The following is a goal for the above program:

$$\mathsf{R} = \{[1, 5], [2, 7]\}\ \&\ \mathsf{is\_rel(R)}\ \&\ \mathsf{dom(R, D)}$$

and the computed solution for D is $\mathsf{D} = \{1, 2\}$. It is important to note that $\mathsf{is\_rel(R)}$ can be used both to test and to compute R; similarly, $\mathsf{dom(R, D)}$ can be used both to compute D from R, and to compute R from D, or simply to test whether the relation represented by dom holds or not.

As can be seen, is_rel and dom are $\{log\}$ implementations of the corresponding concepts in the ZMT. They are part of a $\{log\}$ library implementing almost all the ZMT [7]. Some elements of the ZMT are not supported yet because they are beyond of set theory and, in some cases, would require some extensions to $\{log\}$ or a more complex translation. For example, generic and axiomatic definitions and recursive types are not fully yet supported.

$\{log\}$ has important advantages compared to other Prolog-based tools that can deal with sets. With respect to the simpler scheme which constructs set abstractions on top of an existing Prolog system, typically by using lists (cf. e.g. [21]), $\{log\}$ demonstrates its superiority whenever one has to deal with partially specified sets. For example, even the simple problem of comparing two sets, such as $\{a \mid X\}$ and $\{b \mid Y\}$, may lead to an infinite collection of answers when sets and set operations are implemented using lists, whereas in $\{log\}$ it is dealt as a set unification problem [12] which admits the single more general solution $X = \{b \mid S\}$ and $Y = \{a \mid S\}$ where S is a new variable. Similar considerations hold also when the negative counterparts of the basic set-theoretic operations, such as *'not equal'* and *'not member'*, are taken into account. The use of the list-based implementation of sets, in conjunction with the *Negation as Failure* rule provided by most implementations of Prolog, leads to well-known problems whenever non-ground atoms are involved.

Viewing sets as first-class entities and operations on sets as constraints, as done in $\{log\}$, provides a much more convenient solution. With respect to other proposals which deal with set constraints, such as [13], the main advantage of $\{log\}$ is its generality: sets in $\{log\}$ can contain elements of any type, can be nested and partially specified, whereas sets in the so-called set-constraint languages are restricted to flat sets of known integer elements. Moreover these proposals usually require a finite domain (i.e. a set of sets) is specified for each set variable occurring in a constraint, whereas this is not the case for $\{log\}$. On the other hand, the (incomplete) solvers of the set-constraint languages turn out to be in many cases more efficient than the general solver of $\{log\}$.

$\{log\}$ is fully implemented in Prolog and can be downloaded from [25].

4 Automatic Prototype Generation from Z Specifications

Generating a Prolog prototype from a Z specification involves the automatic translation from Z into Prolog. There are several proposals [8, 31, 3, 17, 14, 29, 33, 18, 37, 15, 23, 35, 7] that confirm that it is possible to implement such a *compiler*. There are also similar works for the B method [34, 1, 16] which defines a mathematical library like the ZMT. However, most of these works are aimed at animating the specification rather than using it as a means to validate user requirements. Then, they do not pay attention to equip the prototype with an acceptable user interface. Animation is thought to be a verification activity carried out by software engineers who want to analyze if a specification verifies some properties; in this sense animation is a complement or an alternative to proof. Instead, we think that if the resulting (Prolog) program is augmented with a GUI, animation becomes rapid prototyping and can be used (also) for functional requirements validation. That is, the prototype can be used by end-users to validate whether or not requirement engineers have understood what the system is supposed to do. Sterling et al. [31] aim at a similar target but they think in a Z-to-Prolog compiler that generates a command-line-like program that users are supposed to play with. We think that this is impractical. Besides, only a handful of these works [1, 23, 16] rely on a powerful, set-oriented constraint language implementing set constraint satisfaction, and even in these cases the underlying mathematics are not described in detail. B-Motion and Brama [28, 19] are two tools based on the B method which go in the same direction proposed here. B-Motion is based on ProB [20]. Both tools seem to be oriented to graphically represent software-controlled physical systems rather than information systems as the one described in the motivation example and the two other examples available on-line [6].

Therefore, we propose to define a simple, GUI specification language (GEL) of which Figure 3 is an example. This language should allow engineers to define the *structure* of a GUI and how it connects with the (Prolog) program generated from the Z specification. This GEL would not need to express the *behavior* of the GUI (for instance, specify when part of the GUI should be disabled).

GUI specification for ERP
Pull-down menu ACCOUNTS has
  Option NEW CLIENT opens ADD NEW CLIENT
  Option INFORM TRANSACTION opens ADD A CREDIT OR DEBIT
  Option ACCOUNTS IN RANGE opens LIST ALL ACCOUNTS IN RANGE
end of menu

. . . . . .

Dialog-box ADD A CREDIT OR DEBIT calls *Transaction* where
  $n?$ is read by dom *clients*
  $n?$ is a `list` labeled as SELECT ID
  $m?$ is a `int` labeled as AMOUNT
end of dialog box

Dialog-box LIST ALL ACCOUNTS IN RANGE calls *AccountsInRange* where
  $a?$ is a `int` labeled as MIN VALUE
  $b?$ is a `int` labeled as MAX VALUE
  $r!$ is a `text` labeled as ACCOUNTS ID IN RANGE
end of dialog box

end of GUI specification

Fig. 3: GUI specification of the Z specification of Figure 1

Then, after a meeting with the client, requirement engineers write Z and GUI specifications and use the prototype generation tool to get the first prototype. In this way, they can go to the next meeting with a prototype over which end-users can validate the requirements [27]. If users propose changes, the Z and GUI specifications are changed accordingly and the tool is run again. This allows an iterative, prototype-based process of requirement elicitation and validation.

The Z specification can be compiled into a Prolog program by a combination between the proposals by Sterling, Ciancarini and Turnidge [31] and by Cristiá, Rossi and Frydman [7]. The main contribution here is to use $\{log\}$ as the constraint solver instead of the list-based implementation of sets proposed by Sterling and his colleagues because this enables the possibility of implementing the SET STATE command discussed in Sect. 5 and the validation method introduced in Sect. 6. Besides, the tool must take into account also the GUI specification accompanying the Z specification. Here XPCE, the SWI-Prolog native GUI library [36], comes handy because it allows a seamless integration of the functional (Z) and the GUI specifications.

The main rules for compiling GUI specifications are the following (follow figures 3 and 4 as an aid to understand these rules):

1. The GUI is implemented inside a XPCE frame object.
2. The menu-bar is implemented as a XPCE menu_bar object.
3. The GUI specification must contain at least one 'pull-down menu' directive.
4. Each 'pull-down menu' directive is compiled as a XPCE popup object.
5. Each 'option' directive in a 'pull-down menu' must open a 'dialog-box'.
6. Each 'option' directive in a 'pull-down menu' is compiled to:

menu_item(option, message(@prolog, dialogBox))

```
initERP :-
    saveState({}, {}), nb_setval(clients, {}), nb_setval(accounts, {}), mainMenu.

mainMenu :-
    Builds a menu bar according to the GUI specification. Each option
    of the pull-down menus calls a predicate like readAccountsInRange below

readAccountsInRange :-
    new(D, dialog('List all accounts in range')),
    send_list(D, append,
              [new(M, int_item(min_value)), new(N, int_item(max_value)),
               text('Accounts ID in range'), new(B, browser),
               button(enter, message(@prolog, accountsInRange, B,
                                     M?selection, N?selection))]),
    send(D, open).

accountsInRange(B, M, N) :- accountsInRangeOk(B, M, N); notARange(B, M, N).

accountsInRangeOk(B, M, N) :-
    b_getval(clients, C), b_getval(accounts, A),
    setlog(M ein int(0, 10000) & N ein int(0, 10000) & M =< N &
           Y = int(M, N) & rres(Y, A, Y1) & dom(Y1, Y2) & set_to_list(Y2, L)),
    writeOutput(B, L), saveNewState(C, A).
```

Fig. 4: Code excerpt of the prototype of the Z specification shown in Figure 1

where dialogBox is a Prolog predicate implementing the dialog-box that the option must open.

7. Each dialog-box is implemented as a XPCE dialog object.
8. Each dialog-box must call an existing Z operation.
9. Each dialog-box must indicate what kind of widget must be used for each and every input and output variable declared in the corresponding Z operation.
10. `text` is implemented as a XPCE text_item object for inputs, and as a browser for outputs; `int` as int_item; `list` as list_browser; etc.
11. A 'is read by' directive in a 'dialog-box' directive simply states that the input variable is read by means of another expression. The selection made through the expression is passed as the value for the input variable.
12. When the enter button in a dialog-box is pressed the following is executed:

    button(enter, message(@prolog, Operation, Input))

    where Operation is a Prolog predicate implementing the corresponding Z operation (this code was generated when the Z specification was compiled) and Input are the input variables waited by the Z operation where each actual value corresponds to one of the values read by the widgets of the dialog-box.

Two more examples of this translation can be downloaded from [6]. They include the Z and GUI specifications and the Prolog+{*log*}+XPCE program.

5 Solving State Predicates for Requirement Validation

In this section we discuss the SET STATE feature presented in the introduction. The inclusion of this feature is based on the following observation: *validating some functional requirements involves taking the prototype to a state such that state variables satisfy a (complex) predicate.* In turn, this can be achieved in three ways: a) use the prototype's GUI to put itself in the desired state; b) write a simple Prolog predicate where each state variable is bound to the desired value and then execute the prototype from this state; and c) write the state predicate, give it to a constraint solver, and use its answer as the prototype's new state.

The first option is annoying, time consuming and error prone; this is the one discussed in [31]. The second one, is error prone because the values may be wrong. It can be improved by: writing the state predicate as in c), proposing a solution to it as in b), and asking the constraint solver whether this solution indeed satisfies the predicate. If it answers "no", try another value. This may take some time but it is safe. However, the best option is c) because the constraint solver does all the work—except proposing the state predicate, see Sect. 6.

The key condition for c) to be feasible is to have a constraint solver capable of solving (complex) Z predicates. Since Z predicates are first-order predicates over the set theory, only a constraint solver for such set theory can deliver the desired results. {*log*} may be such a constraint solver because, according to the results we have obtained when we used it as a test case generator for Fastest [7], it is able to solve many complex Z predicates. In effect, when {*log*} is used as Fastest's test case generator it needs to find solutions to complex Z predicates describing equally complex test conditions. However, not all possible Z predicates can be solved by {*log*} in its current state, since many Z facilities are implemented as user-defined predicates and the {*log*} interpreter can not guarantee efficient processing or even termination for all the considered formulas.

Therefore, the SET STATE menu option of the prototypes generated by our method would wait for either a Z predicate or a {*log*} goal, translate it into {*log*} in the first case, call it and use its answer to set the new state of the prototype. The following considerations must be taken into account:

1. If a Z predicate is entered it would be automatically translated into a {*log*} goal [7]. However, there are cases in which an automatic translation may not yield the best {*log*} code in the sense of {*log*} being able to find a solution for it in a reasonable time. An alternative could be to let users to edit the resulting {*log*} goal to help the tool to find a solution.
2. Typing information about the state variables must be entered along the goal; this has been discussed in [7].
3. Z basic types (i.e. *UID* and *NAME* in Figure 1) pose a problem because their elements have no structure and so {*log*} generates variables instead of constants. This can be solved by post-processing {*log*}'s answer.
4. If some {*log*} goal entered by the user is too complex to be solved, (s)he can write special-purpose {*log*} predicates to help the tool to find a solution.

For example, $\#\,\mathrm{ran}\,accounts > 3$, i.e. the state predicate discussed in the introduction, can be translated as follows:

$$\mathsf{pfunFromRan}(\mathsf{D},\mathsf{R},\mathsf{Accounts})\,\&\tag{1}$$
$$\mathsf{solve}(\mathsf{size}(\mathsf{R},\mathsf{SR})\ \&\ \mathsf{SR}>3)\,\&\tag{2}$$
$$\mathsf{subset}(\mathsf{R},\mathsf{int}(-10000,10000)\ \&\ \mathsf{is\_pfun}(\mathsf{Clients})\,\&\tag{3}$$
$$\mathsf{dom}(\mathsf{Clients},\mathsf{D}).\tag{4}$$

where: (1) builds Accounts from its range, R, and with domain included in D (note that the solution must be a value for Accounts and not only for its range as might be suggested by the original Z predicate); (2) sets the size of R to something greater than 3; (3) gives type information of R and Clients; and (4) is the invariant. $\{log\}$ immediately returns the following:

$$\mathsf{Accounts} = \{[\mathsf{g45},-10000],[\mathsf{g30},-9999],[\mathsf{g15},-9998],[\mathsf{g00},-9997]\}$$
$$\mathsf{Clients} = \{[\mathsf{g45},\mathsf{g630}],[\mathsf{g30},\mathsf{g573}],[\mathsf{g15},\mathsf{g516}],[\mathsf{g00},\mathsf{g459}]\}$$

6 Rigorous Requirement Validation Based on Prototypes

In this section we discuss how scenarios for requirement validation can be automatically generated from the Z specification. These scenarios are a guide for users so they can chose which are worth to be explored and which are not. Each of these scenarios will be given by a Z predicate in terms of the state and input variables of a given Z operation. Then, each scenario will contribute to validate the requirements formalized by the corresponding Z operation. In order to perform the validation, the software engineer must enter each of them in the Set state function discussed in Sect. 5 and then execute the implementation of the corresponding Z operation. End-users must observe the output generated by the prototype and classify it as correct or incorrect.

Our proposal for a rigorous requirement validation process based on prototypes generated from Z specifications (of the requirements themselves) is to apply the Test Template Framework (TTF) [32, 4]. The TTF is a model-based testing (MBT) method tailored to the Z notation. It proceeds by dividing the input space of each Z operation (of a given specification) into so-called *test conditions*. In this context we will use, instead, the term *validation condition*. Each validation condition is given by a Z predicate on the input and state variables of the corresponding Z operation. Each of them describes the conditions for a test case. Precisely, we propose to use these conditions for requirement validation (and, obviously, to later test the implementation). In this sense, validating a requirement consists of:

1. Put the prototype in the state described by the validation condition;
2. Execute the implementation of the Z operation with the input values given by the test condition; and
3. Observe the prototype's behavior to determine if it satisfies end-users.

Fastest is a MBT tool that provides tool support for the TTF [4]. It can automatically generate validation conditions by applying so-called testing tactics. A testing tactic is a systematic and general rule to divide the input space of Z operations. The same testing tactics can be applied for requirement validation. For example, a classical testing tactic is Disjunctive Normal Form (DNF), which rewrites the Z operation in DNF, takes the precondition of each disjunct, and divides its input space with these predicates.

For example, the input space of *Transaction* (Figure 1) is:

$$Transaction^{IS} == \\ [clients : UID \pfun NAME;\ accounts : UID \pfun \mathbb{Z};\ n? : UID;\ m? : \mathbb{Z}]$$

So, when DNF is applied to it the following validation conditions are generated:

$$Transaction_1^{DNF} == [Transaction^{IS} \mid n? \in \mathrm{dom}\ accounts \land m? \neq 0] \\ Transaction_2^{DNF} == [Transaction^{IS} \mid n? \notin \mathrm{dom}\ accounts] \\ Transaction_3^{DNF} == [Transaction^{IS} \mid m? = 0]$$

Therefore, if the prototype is used in each of these conditions, end-users will be validating whether requirement engineers correctly understood (at least part of) what it means to record a transaction of a client. For instance, they will try to record a proper transaction ($Transaction_1^{DNF}$), a transaction of a nonexistent client ($Transaction_2^{DNF}$), etc. As suggested by the TTF, more testing tactics can be applied to further divide the validation conditions obtained so far, thus, getting more revealing ones. Therefore, if engineers and users find that some validation condition should be verified more deeply they can apply more testing tactics to this validation condition to further divide it in more complex cases.

In summary, if Fastest is used to generate the validation conditions, and then each of them is successively entered in SET STATE (cf. Sect. 5), software engineers will have a tool that will assist them in generating and validating a prototype (besides making early progress with testing). {*log*} has already proved to solve many of these conditions [7].

7 Conclusions

We have shown that it would be possible to develop a prototype generator for Z specifications based on the {*log*} constraint solver. These prototypes could include rich GUIs that would make end-users to be able to use them for requirement validation. The TTF could assist software engineers in conducting a systematic requirement validation process, and {*log*} could help them in automating some fundamental activities. This technique would also help during the testing stage. The proposal would have a number of advantages that would pay-off the effort of writing a Z specification.

However, many issues need to be discussed and some problems need to be solved. For example, how prototypes would interface with external systems? How this can be described in Z? Should it be described in Z? Should it be prototyped?

What if a validation condition asks for a state that cannot be reached by a sequence of the operations available in the specification? Should this condition be validated or not? What is a precise characterization of the class of systems within the scope of this proposal? Finding an answer to all these questions, however, seems feasible and is left for future work.

References

1. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B - A constraint solver to animate a B specification. STTT 6(2), 143–157 (2004)
2. Bowen, J.P., Hall, J.A. (eds.): Z User Workshop, Cambridge, UK, 29-30 June 1994, Proceedings. Workshops in Computing, Springer/BCS (1994)
3. Breuer, P.T., Bowen, J.P.: Towards correct executable semantics for Z. In: Bowen and Hall [2], pp. 185–209
4. Cristiá, M., Albertengo, P., Frydman, C.S., Plüss, B., Rodríguez Monetti, P.: Tool support for the Test Template Framework. Softw. Test., Verif. Reliab. 24(1), 3–37 (2014)
5. Cristiá, M., Hollmann, D., Albertengo, P., Frydman, C.S., Monetti, P.R.: A language for test case refinement in the Test Template Framework. In: Qin, S., Qiu, Z. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 6991, pp. 601–616. Springer (2011)
6. Cristiá, M., Rossi, G.: Protoype examples for SETS 2014, `https://www.dropbox.com/s/rgub9d3i10coht8/sets2014-examples.tar.gz`
7. Cristiá, M., Rossi, G., Frydman, C.S.: {log} as a test case generator for the Test Template Framework. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM. Lecture Notes in Computer Science, vol. 8137, pp. 229–243. Springer (2013)
8. Doma, V., Nicholl, R.A.: EZ: A system for automatic prototyping of Z specifications. In: Prehn, S., Toetenel, W.J. (eds.) VDM Europe (1). Lecture Notes in Computer Science, vol. 551, pp. 189–203. Springer (1991)
9. Dovier, A., Omodeo, E.G., Pontelli, E., Rossi, G.: A language for programming in logic with finite sets. J. Log. Program. 28(1), 1–44 (1996)
10. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. 22(5), 861–931 (2000)
11. Dovier, A., Piazza, C., Rossi, G.: A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. ACM Trans. Comput. Log. 9(3) (2008)
12. Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory Pract. Log. Program. 6(6), 645–701 (Nov 2006), `http://dx.doi.org/10.1017/S1471068406002730`
13. Gervet, C.: Conjunto: Constraint propagation over set constraints with finite set domain variables. In: Hentenryck, P.V. (ed.) ICLP. p. 733. MIT Press (1994)
14. Goodman, H.S.: The Z-into-Haskell tool-kit: An illustrative case study. In: Bowen, J.P., Hinchey, M.G. (eds.) ZUM. Lecture Notes in Computer Science, vol. 967, pp. 374–388. Springer (1995)
15. Grieskamp, W.: A computation model for Z based on concurrent constraint resolution. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) ZB. Lecture Notes in Computer Science, vol. 1878, pp. 414–432. Springer (2000)
16. Hallerstede, S., Leuschel, M., Plagge, D.: Validation of formal models by refinement animation. Sci. Comput. Program. 78(3), 272–292 (2013)
17. Hasselbring, W.: Animation of Object-Z specifications with a set-oriented prototyping language. In: Bowen and Hall [2], pp. 337–356

18. Hewitt, M.A., O'Halloran, C., Sennett, C.T.: Experiences with PiZA, an animator for Z. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM. Lecture Notes in Computer Science, vol. 1212, pp. 37–51. Springer (1997)
19. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-Motion Studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 5825, pp. 202–204. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04570-7_17
20. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Keijiro, A., Gnesi, S., Mandrioli, D. (eds.) FME. Lecture Notes in Computer Science, vol. 2805, pp. 855–874. Springer-Verlag (2003)
21. Munakata, T.: Notes on implementing sets in prolog. Commun. ACM 35(3), 112–120 (Mar 1992), http://doi.acm.org/10.1145/131295.131300
22. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and CLP with sets. In: PPDP. pp. 219–229. ACM (2003)
23. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods. Lecture Notes in Computer Science, vol. 4591, pp. 480–500. Springer-Verlag (2007)
24. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
25. Rossi, G.: {*log*} (2008), http://www.math.unipr.it/~gianfr/setlog.Home.html, last access: December 2013
26. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Tech. rep., ORA Canada (1997)
27. Schrage, M.: Never go to a client meeting without a prototype. IEEE Software 21(2), 42–45 (2004)
28. Servat, T.: BRAMA: A new graphic animation tool for B models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007: Formal Specification and Development in B, Lecture Notes in Computer Science, vol. 4355, pp. 274–276. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11955757_28
29. Sherrell, L.B., Carver, D.L.: FunZ: An intermediate specification language. Comput. J. 38(3), 193–206 (1995)
30. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)
31. Sterling, L., Ciancarini, P., Turnidge, T.: On the animation of "not executable" specifications by Prolog. International Journal of Software Engineering and Knowledge Engineering 6(1), 63–87 (1996)
32. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering 22(11), 777–793 (Nov 1996)
33. Valentine, S.H.: The programming language Z−. Information & Software Technology 37(5-6), 293–301 (1995)
34. Waeselynck, H., Behnia, S.: B model animation for external verification. In: ICFEM. pp. 36–45 (1998)
35. West, M.M.: The use of a logic programming language in the animation of Z specifications. In: Dahl, V., Niemelä, I. (eds.) ICLP. Lecture Notes in Computer Science, vol. 4670, pp. 451–452. Springer (2007)
36. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP 12(1-2), 67–96 (2012)
37. Winikoff, M., Dart, P., Kazmierczak, E.: Rapid prototyping using formal specifications. In: In Proceedings of the 21st Australasian Computer Science Conference. pp. 279–294. Springer-Verlag (1998)