ZLOG: A Prolog Toolkit for Prototyping Z Specifications

Girish Keshav Palshikar

Tata Research Development and Design Centre,
#54B, Hadapsar Industrial Estate,
Pune 411013, India.
Tel.: +91 20 671058
Email: girishp@pune.tcs.co.in

Abstract

Z is a popular notation for formal specification of software systems. It is based on set theory and typed first order predicate logic and provides the fundamental mathematical structures (with operations upon them) like sequences, sets, relations and functions. Z includes the schema notation for structuring and incremental development of reusable specifications. Formal specifications in Z are usually not directly executable as prototypes. We describe ZLOG - a Prolog toolkit to dynamically manipulate the mathematical objects in Z. ZLOG specifications can be incrementally developed, tested, queried, analyzed for properties and executed as prototypes. ZLOG helps to interactively improve problem understanding and detect at the earliest functional shortcomings - a frequent reason causing delayed client acceptance and large overheads in rework.

Keywords: Formal Specifications; Z; Prolog; Set manipulation; Symbolic computation

1. Introduction

Essentially, formal specifications are a precise mathematical expression of the desired functionality, behaviour and properties of a system against which the design and implementation can be cross-referenced and validated. The activity of developing formal specifications of software systems is increasingly being recognized as important for improving quality and cutting software development costs. Formal specifications bring clarity and un-ambiguity to the statement of system requirements. Any problems in the requirements can be detected early in the software life cycle. Formal specifications also serve as an unambiguous means of communication among the various agencies involved in the complete system development cycle.

Several notations, based on different mathematical frameworks, have been developed for writing formal specification of system requirements. Sets, relations, functions, sequences and partial orders are fundamental mathematical structures as well as important data types in terms of which solutions of many computational problems can be expressed. Mixing these mathematical structures with typed first order predicate logic leads to a rich notation, called Z [1, 2, 3, 4] to express formal specifications of complex software systems. In addition, Z also provides the schema calculus, which helps in organizing the formal specifications of a system in a modular and hierarchical manner and also helps to improve the reusability of the specification components. Z has been widely used as a formal specification notation in various real-life industrial projects.

During and after the development of formal specifications, there is a need to check the specifications for correctness and completeness [5]. Theorem-proving tools can be used for the verification of properties and consequences of the formal specifications. Alternatively, the specifications themselves
can be executed and tested against typical use cases. This testing is aimed at finding out any errors and functional shortcoming in the specifications. Executing the specifications can also be looked upon as a very high-level prototype of the desired system. Specification execution also helps to check the properties of the system and to uncover its departures from the expected behaviour.

However, Z is an abstract specification notation without any direct relationship with a model of execution. Since Z allows working with infinite sets and partial specifications, it is possible to write Z specifications that cannot be executed directly. However, the Z notation is sufficiently “close” to logic and functional programming languages, so that a number of attempts have been made to (manually) translate Z specifications to such languages for execution.

West and Eaglestone [6] discuss two approaches to animate Z specifications using Prolog: (a) formal program synthesis and (b) structure simulation. They argue about the limitations of the first approach, and recommend the later approach, which is also the basis of the work in this paper. However, they do not describe a complete tool to do a faithful animation of Z facilities. In [7], a similar approach is followed, except that it describes a (manual) translation of Z specifications into the functional programming language Haskell. Rudimentary set management predicates are discussed in [8], [9], and [10]. However, they are insufficient to be directly usable for prototyping Z specifications since they lack facilities for implicit set definitions, sets of sets as well as specific Z operators to manipulate these objects. In [11], the theoretical relationships between logic programming and logical specifications are discussed.

In this paper, we demonstrate that by imposing the following simple structural restrictions on Z specifications, we can bring the Z specifications “sufficiently” close to Prolog’s execution model.

- Given (i.e., unspecified) sets are not permitted; all sets must be defined.
- Infinite sets (like \( \mathbb{N} \)) are not permitted; all sets must be finite.
- There must be a unique common data state-space that is being manipulated by all schema.

In addition, the logical conditions in the Z specifications may be written with a more “operational” mindset so as to keep the translation to ZLOG easier.

We describe a tool called ZLOG, which can be used to prototype and interactively execute formal specifications written Z. Specifications written in Z need to be manually translated into executable ZLOG programs. ZLOG is essentially a portable Prolog toolkit for uniform definition and symbolic manipulation of mathematical types (e.g., sets, relations, functions, sequences etc.) of Z in the logic programming framework of Prolog. ZLOG also provides dynamic type checking and operators in the Z schema calculus. ZLOG supports both explicit and implicit definitions of sets and ZLOG operators work for both representations. ZLOG provides a uniform treatment of the mathematical objects in Z; e.g., a function can also be manipulated as a relation or as a set as necessary.

In this paper, we assume a basic familiarity with the Z notation as well as with Prolog. The rest of the paper is organized as follows. In the next section, we discuss and illustrate the ZLOG facilities for symbolic manipulation of mathematical objects in Z. We then demonstrate, with an example, the process of the conversion of a Z specification into ZLOG program and executing it as a prototype. We conclude with an outline of further work.
2. The ZLOG Toolkit

2.1 Sets In ZLOG

In ZLOG, a set is defined when a way has been provided to enumerate all of its elements. ZLOG deals with finite sets only; hence, one cannot define integer or real as a set in ZLOG. In ZLOG, there are two ways of defining a set: (a) explicit definition (b) implicit definition.

In explicit set enumeration, a set name is associated with an explicit listing of all the elements in the set. The simplest way to list the elements in a set is to provide them as an unordered collection of elements enclosed in curly braces { and }. An explicit enumeration can also be constructed from other, previously defined, sets using the usual set operators. The general syntax of an explicit enumeration of a set in ZLOG is as follows:

\[
\text{set SetName ::= SetExpression.}
\]

where, SetName is a Prolog ground term giving a name for the set, SetExpression is a usual set expression made up of previously defined sets and the usual set operators and ::= is the set enumeration operator. The left hand side of ::= must always be a unary Prolog term called set and the lone argument of this term must be a Prolog atom giving the name of the set. Thus, in ZLOG, an explicitly enumerated set must always have a name.

Examples of explicit enumeration of a number of sets in ZLOG syntax are given below in Figure 1. An explanatory comment is also given next to each explicit enumeration. There is no restriction on the nature of the elements in a set; a set element can be any valid Prolog term, including another set. Note that the set s5 is explicitly defined to be the empty set and the set s7 is a collection i.e., it contains other sets as elements. Notice how the unary Prolog term set s1 can be used in a set expression for the previously explicitly enumerated set s1.

\[
\begin{align*}
\text{set s1 ::= \{ a,b,c \}.} & \quad \% \text{Set has name s1 and elements \{a,b,c\}} \\
\text{set s2 ::= \{ c,d \} union set s1.} & \quad \% \text{Set has name s2 and elements \{a,b,c,d\}} \\
\text{set s3 ::= \{ b,d,e \} intersection set s2.} & \quad \% \text{Set has name s3 and elements \{b,d\}} \\
\text{set s4 ::= \{a,b\} union \{b,c,d\} intersection set s3.} & \quad \% \text{Set s4 has elements \{a,b,d\}} \\
\text{set s5 ::= \{\}.} & \quad \% \text{Set has name s5 and is an empty set} \\
\text{set s6 ::= set s1 product \{1,2\}.} & \quad \% \text{Set s6 is Cartesian product of s1 and \{1,2\}} \\
\text{set s7 ::= \{set s1,\{c,d\}\}.} & \quad \% \text{Collection s7 contains two set-elements \{a,b,c\} and \{c,d\}} \\
\text{set s8 ::= \{b,c,d,e\} difference set s1.} & \quad \% \text{Set has name s8 and elements \{d,e\}} \\
\text{set s9 ::= set s1 product \{1,2\} product \{x,y\}.} & \quad \% \text{Set s6 = s1 × \{1,2\} × \{x,y\}}
\end{align*}
\]

Figure 1. Examples of explicit enumeration of sets in ZLOG.

ZLOG contains a number of set manipulation predicates (Appendix 1). The following are some examples to illustrate the use of the various set manipulation predicates in ZLOG. Note that set expressions can be used wherever a set can appear. The Cartesian product is generated in terms of tuples. In ZLOG, a tuple is a Prolog term with a reserved name t and enclosed in ( and ); e.g., \(t(a,b,c)\) and \(t(a,b), [a,b]\) are ZLOG tuples. The number of arguments in a tuple term is called its tuple arity; e.g., arity of \(t(a,b,c)\) is 3. The predicate add_to adds the given element to the given set and has the format Element add_to Set. The ZLOG set management library contains many other predicates; e.g. for partitioning, cover, minterms and so on.

\[- \{c,b\} \text{ belongs_to } \{a,\{b,c\},d\}. \% \text{given element \{c,b\} is matched with element \{b,c\}} \]
\[\text{yes}\]
?- X belongs_to {a,\{b,c\},d}. % successively returns elements of the set \{a,\{b,c\},d\}
X = a ;
X = \{b,c\} ;
X = d ;
no

?- \{a,\{b,c\},d\} seteq \{d,a,\{c,b\}\}. % given sets are same even when order of elements is different
yes

?- \{a,\{b,c\},d\} seteq X. % seteq can also be used for instantiating a set variable
X = \{a,\{b,c\},d\}

?- N cardinality \{a,\{b,c\},d\}. % get the cardinality of the given set
N = 3

?- \{\{c,b\},a\} subset \{a,\{b,c\},d\}. % subset works even when order of elements is different
yes

?- X seteq \{a,\{b,c\},d\} union \{p,\{c,b\}\}. % seteq instantiates X after evaluating given set expression
X = \{a,\{b,c\},d,p\}

?- X seteq \{a,\{b,c\},d\} intersection \{p,\{c,b\}\}.
X = \{\{b,c\}\}

?- X seteq \{a,\{b,c\},d\} difference \{p,\{c,b\}\}.
X = \{a,d\}

?- X seteq \{a,\{b,c\},d\} sym_difference \{\{c,b\},d,e\}.
X = \{a,e\}

?- X seteq \{a,\{b,c\}\} product \{1,2\}. % seteq instantiates X to Cartesian product of given 2 sets
X = \{t(a,1),t(a,2),t(\{b,c\},1),t(\{b,c\},2)\}

?- X seteq \{a,\{b\}\} product \{x\} product \{1,2\}. % seteq instantiates X to product of given 3 sets
X = \{t(a,x,1),t(a,x,2),t(b,x,1),t(b,x,2)\}

?- set sl := set sl union \{c,d,e\}. % update set sl to contain new elements d and e
yes

?- X seteq set sl. % verify that sl = \{a,b,c,d,e\}
X = \{c,b,a,d,e\}

A set can also be defined implicitly by means of a predicate; e.g., states = \{x \mid x is a state in USA\}. Implicit definitions are storage efficient, since all elements of the sets are not explicitly stored. They are also compact and easier to understand and manipulate. They are especially suitable when the elements of a set change dynamically. Other methods of defining sets [10] can be easily incorporated in ZLOG.

In ZLOG, a set can be defined implicitly by means of a predicate having the name set and arity of at least two. The first argument of a set predicate must always be a Prolog atom giving the name of the set. The second argument of set refers to an element of the set. The set defining predicate set must return the successive set elements in its second argument upon backtracking. It must be complete in the sense that it must generate all members. A set-defining set predicate must ensure that duplicate elements are not generated during backtracking. Other arguments of the set predicate are optional and may be application dependent. A set defining set predicate must be correct i.e. it must ensure that
given element is correctly recognized as a member. A set defining set predicate must fail when there are no more elements in the set.

For example, the predicate large_cities defines a set of cities having population at least as much as the given value. The following dialog shows that large_cities satisfies all the above conditions and hence is an acceptable set-defining predicate. It is the programmer's responsibility to ensure that the set-defining predicate meets all these conditions and terminates by failure when there are no more elements (i.e. defines a finite set).

city(london,uk,6700).
city(new_york,usa,7100).
city(washington,usa,600).

set large_cities(X,MinPop) :- city(X,_,Pop), Pop >= MinPop.

?- X belongs_to set large_cities(_,5000).
X=london ;
X=new_york ;
no

All the set management predicates defined in the last section can be used on implicitly defined sets. Any output sets generated by these predicates are usually in explicit list form only. For example,

?- set large_cities(_,5000) seteq {new_york,london}.
yes

?- 3 cardinality set large_cities(_,100).
yes

?- set large_cities(_,5000) subset set large_cities(_,100).
yes

?- X seteq large_cities(_,5000) intersection {london,washington}.
X = {london}

?- X seteq {london,\{b,c\},d} sym_difference large_cities(_,5000).
X = {\{b,c\},d,new_york}

?- X seteq large_cities(_,5000) product \{warm, cold\}.
X = {\{new_york,cold\},\{new_york,warm\},\{london,cold\},\{london,warm\}}

2.2 Relations In ZLOG

A binary relation \( R \) between sets \( X \) and \( Y \) is a subset of the Cartesian product of \( X \) and \( Y \). Since binary relations are of major interest, in this section, we primarily illustrate the ZLOG predicates for management of binary relations. In ZLOG, a binary relation is represented as an explicitly or implicitly defined set of tuples of arity 2. All relation management predicates in ZLOG work with both explicitly and implicitly defined relations. All set management predicates in ZLOG also work with relations, since relations are nothing but sets of tuples.

We shall now define a relation used as an example. Let \( I_7 = \{1,2,3,4,5,6,7\} \) be a set. For \( X,Y \) in \( I_7 \), \( X \text{ div3 } Y \) if \( X-Y \) is divisible by 3.

set i7 ::= \{1,2,3,4,5,6,7\}.
set div3(t(X,Y)) :-
    X belongs_to set i7,
    Y belongs_to set i7,
    0 is (X - Y) mod 3.
The predicate `isrel2(+Rel2)` checks if a given set `Rel2` defines a valid binary relation or not i.e. whether it consists only of ordered tuples or not. This predicate succeeds if the given relation `Rel2` (in explicit or implicit form) is a set consisting of only tuples of same arity. The set predicate `empty(+Rel2)` succeeds if given relation `Rel2` is an empty set. The predicate `identity(+Rel2)` is true if given binary relation `Rel2` is an identity relation.

```
?- isrel2([a,t(a,1),t(a,2)]).
no
?- isrel2(set div3(_)).
yes
?- empty({}).
yes
?- identity([t(a,a),t(b,b)]).
yes
```

The set expression `dom +Rel2 (ran +Rel2)` stands for the set, which is the domain (range) of the given binary relation `Rel2`.

```
?- X seteq dom set div3(_).
X = {1,2,3,4,5,7}
?- set i7 seteq ran set div3(_).
yes
```

The set expression `+U dom_restrict +Rel2` stands for the binary relation which consists of only those tuples `t(X,Y)` in the given binary relation `Rel2` such that `X` is in the given set `U`. The set expression `+U dom_antirestrict +Rel2` stands for the binary relation which consists of only those tuples `t(X,Y)` in the given binary relation `Rel2` such that `X` is not in the given set `U`.

```
?- X seteq {1,2,3} dom_restrict set div3(_).
X = {t(3,6),t(3,3),t(2,5),t(2,2),t(1,7),t(1,4),t(1,1)}
?- X seteq {1,2,3} dom_antirestrict set div3(_).
X = {t(7,7),t(7,4),t(7,1),t(6,6),t(6,3),t(5,5),t(5,2),t(4,7),t(4,4),t(4,1)}
```

The set expression `+U ran_restrict +Rel2` stands for the binary relation which consists of only those tuples `t(X,Y)` in the given binary relation `Rel2` such that `Y` is in the given set `U`. The set expression `+U ran_antirestrict +Rel2` stands for the binary relation which consists of only those tuples `t(X,Y)` in the given binary relation `Rel2` such that `Y` is not in the given set `U`.

```
?- X seteq {1,2,3} ran_restrict set div3(_).
X = {t(6,3),t(3,3),t(5,2),t(2,2),t(7,1),t(4,1),t(1,1)}
?- X seteq {1,2,3} ran_antirestrict set div3(_).
X = {t(7,7),t(7,4),t(6,6),t(5,5),t(4,7),t(4,4),t(3,6),t(2,5),t(1,7),t(1,4)}
```

The predicate `from_to(+Rel2,+DSet,+RSet)` succeeds if the domain of given relation `Rel2` is a subset of given set `DSet` and its range is a subset of given set `RSet`. The predicate `rel2_in(+Rel2,+Set)` succeeds if the domain and range of given relation `Rel2` are both subsets of the given set `Set`. The predicate `total(+Rel2,+DSet)` succeeds if the given binary relation `Rel2` is defined for all elements of `DSet` i.e. for all `X` in `DSet` there is at least one tuple `t(X,_)` in `Rel2`.

```
?- from_to(set div3(_), set i7, [1,2,3,4,5,6,7]).
yes
?- rel2_in(set div3(_), set i7).
yes
?- total(set div3(_),[1,2,3,4,5,6,7]).
yes
```
The predicate reflexive(+Rel2,+Set) succeeds if the given binary relation Rel2 in given set Set is reflexive. The predicate reflexive(+Rel2) succeeds if the given total binary relation Rel2 is reflexive. Similarly, the predicate irreflexive(+Rel2,+Set) succeeds if the given binary relation Rel2 in given set Set is irreflexive. The predicate irreflexive(+Rel2) succeeds if the given total binary relation Rel2 is irreflexive. The predicate symmetric(+Rel2) succeeds if given binary relation Rel2 is symmetric. The predicate transitive(+Rel2) succeeds if given binary relation Rel2 is transitive. The predicate antisymmetric(+Rel2) succeeds if given binary relation Rel2 is antisymmetric. The predicate equivalence(+Rel2,+Set) succeeds if the given binary relation Rel2 in given set Set is an equivalence relation. The predicate equivalence(+Rel2) succeeds if the given total binary relation Rel2 is an equivalence relation.

?- reflexive(set div3(_),set i7).
  yes
?- reflexive(set div3(_)).
  yes
?- symmetric(set div3(_)).
  yes
?- transitive(set div3(_)).
  yes
?- antisymmetric(set div3(_)).
  no
?- equivalence(set div3(_),{1,2,3,4,5,6,7}).
  yes

The set expression +Rel1 compose +Rel2 stands for the binary relation obtained as composition of the given two binary relations Rel1 and Rel2. The binary relation obtained by the composition of the two given binary relations Rel1 and Rel2 consists of all those tuples t(X,Z) such that t(X,Y) belongs to Rel1 and t(Y,Z) belongs to Rel2, for some Y in the range of Rel1 and domain of Rel2.

?- X seteq {t(1,2),t(3,4),t(2,2)} compose {t(4,2),t(2,5),t(3,1)}.
  X = {t(1,5),t(3,2),t(2,5)}

The set expression inverse +Rel2 stands for the binary relation obtained by taking the converse (or inverse) of the given binary relation Rel2. The binary relation obtained by the inverse of the given binary relation Rel2 consists of all those tuples t(Y,X) such that t(X,Y) belongs to Rel2.

?- X belongs_to inverse {t(1,2),t(3,4),t(2,2)}.
  X = t(2,1); X = t(4,3); X = t(2,2); no

The set expression power(+Rel2,+N) stands for the binary relation obtained by taking the Nth power of the given binary relation Rel2. The index N must be a nonnegative integer (N \geq 0). The 0th power of a binary relation is the empty set. The first power of a binary relation is the set itself. The Nth power (N > 1) of a binary relation is the composition of the relation with the N-1th power of the relation.

?- X seteq power({t(a,b),t(b,c),t(c,a)},2).
  X = {t(a,c),t(b,a),t(c,b)}
?- X seteq power({t(a,b),t(b,c),t(c,a)},3).
  X = {t(a,a),t(b,b),t(c,c)}

The set expression transitive_closure Rel2 stands for the binary relation, which is the transitive closure of the given binary relation Rel2.
?‑ X seteq transitive_closure {t(a,b),t(b,c),t(c,a)}.
X = {t(a,a), t(b,b), t(c,c), t(a,c), t(b,a), t(c,b), t(a,b), t(b,c), t(c,a)}

2.3 Functions In ZLOG

A function f between sets X and Y is binary relation from X to Y such that for every x ∈ X there is a unique y ∈ Y with t(x,y) ∈ f. In ZLOG, a function is represented as an explicitly or implicitly defined relation (i.e., a set of tuples of arity 2) with the restriction that an element of the domain is related to at most one element in the range. All function management predicates in ZLOG work with both explicitly and implicitly defined functions. Moreover, as functions are relations and sets, all relation and set management predicates in ZLOG also work with functions.

The predicate isfunc(+Func) succeeds if the given relation defines a function. The predicate func_partial(+Func,+DomainSet) succeeds if the given binary relation Func is a valid partial function for the given DomainSet. The predicate func_total(+Func,+DomainSet) succeeds if the given binary relation Func is a valid total function for the given DomainSet.

?- isfunc({t(a,0),t(b,1),t(a,2)}).
no
?- func_partial({t(a,0),t(b,1)},{a,b,c}).
yes
?- func_total({t(a,0),t(b,1)},{a,b,c}).
no
?- func_total({t(a,0),t(b,1),t(c,0)},{a,b,c}).
yes

The predicate onto(+Func,+RangeSet) succeeds if the given function Func is onto the given RangeSet. The predicate one_to_one(+Func) succeeds if the given function Func is one-to-one. The predicate one_to_one_onto(+Func,+RangeSet) succeeds if the given function Func is one-to-one and onto the given RangeSet.

?- onto({t(a,0),t(b,1),t(c,0)},{0,1}).
yes
?- one_to_one({t(a,0),t(b,1),t(c,0)}).
no
?- one_to_one_onto({t(a,0),t(b,1),t(c,2)}).
yes

The set expression +Func1 override +Func2 stands for the function obtained by overriding the given function Func1 with the given function Func2. Essentially, the set denoted by this set expression consists of those tuples t(X,Y) such that either (i) t(X,Y) belongs to Func1 and no tuple t(X,_) belongs to Func2; or (ii) t(X,Y) belongs to Func2 and no tuple t(X,_) belongs to Func1; or (iii) t(X,Y) belongs to both Func1 and Func2.

?- X seteq {t(mary,19),t(john,23)} override {t(john,25),t(george,30)}.
X = {t(mary,19), t(john,25), t(george,30)}

Other relation set and relation operators can also be applied to functions. For example,

?- X seteq {john,mary} dom_antirestrict {t(mary,19),t(john,23),t(george,30)}.
X = {t(george,30)}
3. Prototyping Z Specifications in ZLOG

We now illustrate the process of converting Z specifications to ZLOG program and executing it to prototype the specifications. Consider a simple system [4], which maintains the directory of the internal telephone numbers (extensions numbers) for the members of an organisation. We first present the Z specification for this internal telephone directory (ITD) system and then illustrate how Z specification can be prototyped using ZLOG.

3.1 ITD: Z Specifications

The given sets PERSON and PHONE denote the set of all possible persons and telephone numbers respectively. The set REPORT lists the responses indicating results for manipulation of data.

[PERSON] -- the set of all possible persons
[PHONE] -- the set of all possible telephone numbers

REPORT::= ok | 'not a member' | 'entry already exists' | 'unknown name'
| 'unknown number' | 'unknown entry' | 'already a member'

The global data-state of the system is defined by the schema PhoneDB. The set members indicates the persons who are currently members of the organisation. The relation telephones indicates the current persons who have been allocated a telephone. Note that a member can have zero, one or more telephones; also, a given telephone number may be associated with zero, one or more members. Note also, that only members can have telephones.

PhoneDB

| members : F PERSON |
| telephones : PERSON ↔ PHONE |

\(\text{dom telephones} \subseteq \text{members} \quad \text{-- only members can have phones}\)

The initial value of the data-state is given by the schema InitPhoneDB. Clearly, the initial data-state exists and is unique.

InitPhoneDB =\text{def} [ \text{PhoneDB} \mid \text{members} = {} \land \text{telephones} = {} ]

The schema AddEntry defines the operation, which associates a telephone with a person who is already a member of the organisation. The schema ErrorNotMember reports an error if the given person is not a member of the organisation. The schema ErrorEntryAlreadyExists reports an error if the given person is already having the given telephone. The schema Success reports the successful completion of an operation. The complete specification of the operation to associate a telephone with a member is given by the schema DoAddEntry.

AddEntry

\(\Delta \text{PhoneDB}\)

name?: PERSON

new?: PHONE

name? \in \text{members}

(name?, new?) \notin \text{telephones}

\[\text{telephones'} = \text{telephones} \cup \{ (name?, new?) \}\]

\[\text{members'} = \text{members}\]
ErrorNotMember

= PhoneDB
name? : PERSON
rep! : REPORT

name? \notin members
rep! = 'not a member'

ErrorEntryAlreadyExists

= PhoneDB
name? : PERSON
new? : PHONE
rep! : REPORT

(name?, new?) \in telephones
rep! = 'entry already exists'

Success = \text{def} \ [ \ rep! : REPORT | rep! = \text{ok} ]

DoAddEntry = \text{def} (AddEntry \land Success) \lor ErrorNotMember \lor ErrorEntryAlreadyExists

The schema FindPhones defines the operation, which returns all the telephone numbers associated with the given person. The schema ErrorUnknownName reports an error if currently no telephone is associated with the given person. The complete specification of the operation to interrogate the current data-state for all telephones associated with a given member is given by the schema DoFindPhones. This operation does not alter the contents of the current data-state.

FindPhones

= PhoneDB
name? : PERSON
nums! : F PHONE

name? \in \text{dom telephones}
nums! =

\text{telephones} (\mid \{\text{name}\} \mid)

ErrorUnknownName

= PhoneDB
name? : PERSON
rep! : REPORT

name? \notin \text{dom telephones}
rep! = 'unknown name'

DoFindPhones = \text{def} (FindPhones \land Success) \lor ErrorUnknownName

The schema FindNames defines the operation which returns all the members associated with the given telephone number. Note the use of the relational inversion operator \sim. The schema ErrorUnknownNumber reports an error if currently no name is associated with the given telephone number. The complete specification of the operation to interrogate the current data-state for all persons associated with a given telephone number is given by the schema DoFindNames. This operation does not alter the contents of the current data-state.

FindNames

= PhoneDB
num? : PHONE
names!: F PERSON

num? \in \text{ran telephones}
nums! =

\text{telephones}^{-1} (\mid \{\text{name}\} \mid)

ErrorUnknownNumber

= PhoneDB
num? : PHONE
rep! : REPORT

num? \notin \text{ran telephones}
rep! = 'unknown name'

DoFindNames = \text{def} (FindNames \land Success) \lor ErrorUnknownNumber

The schema RemoveEntry defines the operation, which removes the given number associated with the given member from the current data-state. The schema ErrorUnknownEntry reports an error if currently the given person does not have the given telephone number. The complete specification of the operation to remove an entry from the data-state is given by the schema DoRemoveEntry.
RemoveEntry
\[ \Delta \text{PhoneDB} \]
\[
\begin{align*}
\text{name?} & : \text{PERSON} \\
\text{old?} & : \text{PHONE} \\
\end{align*}
\]
\[
\begin{align*}
\text{(name?,old?)} & \in \text{telephones} \\
\text{telephones'} & = \text{telephones} \setminus \{ (\text{name?,old?}) \}
\end{align*}
\]

ErrorUnknownEntry
\[ = \text{PhoneDB} \]
\[
\begin{align*}
\text{name?} & : \text{PERSON} \\
\text{old?} & : \text{PHONE} \\
\text{rep!} & : \text{REPORT} \\
\end{align*}
\]
\[
\begin{align*}
\text{(name?,old?)} & \notin \text{telephones} \\
\text{rep!} & = \text{‘unknown entry’}
\end{align*}
\]

\[ \text{DoRemoveEntry} \overset{\text{def}}{=} (\text{RemoveEntry} \land \text{Success}) \lor \text{ErrorUnknownEntry} \]

The schema \text{RemoveMember} defines the operation, which removes the given member from the current data-state. The given person is deleted from the set \text{members} and all telephones associated with him are removed from the relation \text{telephones}. The schema \text{ErrorUnknownEntry} reports an error if currently the given person does not have the given telephone number. The complete specification of the operation to remove a member from the data-state is given by the schema \text{DoRemoveMember}.

AddMember
\[ \Delta \text{PhoneDB} \]
\[
\begin{align*}
\text{name?} & : \text{PERSON} \\
\text{members'} & = \text{members} \cup \{ \text{name?} \} \\
\text{telephones'} & = \text{telephones}
\end{align*}
\]

\[ \text{DoRemoveMember} \overset{\text{def}}{=} (\text{RemoveMember} \land \text{Success}) \lor \text{ErrorNotMember} \]

The schema \text{AddMember} defines the operation, which adds a new member of the organisation to the current data-state. The given person is added to the set \text{members} and no telephone is initially associated with him. The schema \text{ErrorAlreadyMember} reports an error if currently the given person is already a member of the organisation. The complete specification of the operation to add a member to the data-state is given by the schema \text{DoAddMember}.
ErrorAlreadyMember

\(=\) PhoneDB
defname? : PERSON
defrep! : REPORT

defname? \(\in\) members
defrep! = 'already a member'

\[\text{DoAddMember} =_{\text{def}} (\text{AddMember} \land \text{Success}) \lor \text{ErrorAlreadyMember}\]

3.2 ITD: ZLOG Program

Before we describe the ZLOG program for the ITD specifications in Z, we need to understand some basic concepts in ZLOG: variable lists, definition of a schema predicate as a standardized Prolog clause, invoking a schema predicate using \(z\text{call}\). A ZLOG program is a normal Prolog program, except that it contains specific schema predicates and makes use of the set operators. A ZLOG program is manually derived from the given Z specifications.

ZLOG has the concept of a variable list. A variable list is a Prolog list and each element in this list is has the following form: \(\text{VariableName} : \text{Type}\), where \(\text{Type}\) is any valid set expression giving the type of the variable which has the name \(\text{VariableName}\). \(\text{VariableName}\) is a genuine Prolog variable and so \(\text{VariableName}\) should begin with a capital letter. The value (instantiation) of \(\text{VariableName}\) should be an element of the set given by the set expression \(\text{Type}\). For example, the variable list \([A : \text{set 'PERSON'}, B : \text{set integer}]\) says that the Prolog variable \(A\) can be instantiated with only an element of the ZLOG defined set \(\text{set 'PERSON'}\) and that the variable \(B\) can be instantiated with only an integer value. The order of the elements in a variable list is important.

Each Z schema is modeled by a Prolog schema predicate having the same name and three arguments. The first argument is the list of schema predicates corresponding to the schema included in the current schema. The second and third arguments respectively are the variable lists corresponding to the input and output arguments for the schema. The schema predicates declared in the first argument are called both before and after the body of a schema predicate is executed.

The ZLOG predicate \(z\text{call} (\text{SchemaPredicate})\) should be used to call a schema predicate (rather than the built-in \(\text{call}\) predicate in Prolog). When \(z\text{call}\) is used to call a schema predicate, there is no need to pass the first argument to it. It is important to remember this distinction: when a schema predicate is defined as a Prolog clause, it has three arguments, however, it is called (using \(z\text{call}\)), with only two arguments.

The complete ZLOG program obtained from the Z specifications in the previous section is given in Appendix B. We now describe only some important parts of this ZLOG program. The given sets \(\text{PERSON}, \text{TELEPHONE}\) and \(\text{REPORT}\) are now be explicitly defined; we define the sets \(\text{PERSON}\) and \(\text{TELEPHONE}\) to contain some arbitrary elements.

\[\text{set 'PERSON'} ::= \{a,b,c,d,e\}\]
\[\text{set 'PHONE'} ::= \{1,2,3,4,5,6\}\]
\[\text{set 'REPORT'} ::= \{ \text{ok, 'not a member', 'entry already exists', 'unknown name', 'unknown number', 'unknown entry', 'already a member'} \} .\]

We assume that the ITD system maintains the data declared in schema \(\text{PhoneDB}\) as a global data. The variables \(\text{members} : F \text{PERSON}\) and \(\text{telephones} : \text{PERSON} \leftrightarrow \text{TELEPHONE}\) are represented by two ZLOG sets: set \(\text{members}(_){}\) and set \(\text{telephones}(_){}\). Both the sets are globally available to the ZLOG program. Thus the system-state at any point during the prototyping is
defined by the elements currently contained in these two sets. Initially, there are no elements in these two sets. The schema PhoneDB, which sets the preconditions on the system-state, is modeled by the Prolog predicate schema 'PhoneDB'. The three empty lists (given as arguments to the schema predicate 'PhoneDB') correspond to the fact that the schema PhoneDB does not include any other schema and does not declare any input or output variables.

'Schema PhoneDB'([],[],[]):
    dom set telephones(_) subset set members(_).

A call to the schema predicate, made as zcall('PhoneDB'([],[])), verifies whether the current system-state satisfies the required constraint that only members can have telephones.

The schema AddEntry is defined by the ZLOG schema predicate 'AddEntry'. Schema AddEntry includes the schema PhoneDB. Therefore, the first argument of the schema predicate 'AddEntry' is a singleton list, containing only one schema predicate 'PhoneDB'([],[]), which is in the format required by zcall. The second argument to the predicate schema 'AddEntry' declares two input variables Name and New, along with their types. The schema AddEntry has no output variables; hence, the third argument to the predicate schema 'AddEntry' is an empty variable list. These input and output variables are Prolog variables and hence their names begin with a capital letter.

The body of the schema predicate 'AddEntry' closely corresponds to the predicate part of schema AddEntry and makes use of the ZLOG operators for set manipulation. The update of the state variable set telephones(_) is achieved by a call to the ZLOG set update operator :=. The schema predicate 'PhoneDB'([],[]), declared in the first argument, is automatically called both before and after the body of the schema predicate 'AddEntry' is executed.

'Schema AddEntry'(['PhoneDB'([],[])],
    [Name : set 'PERSON', New : set 'PHONE'], []):
    Name belongs_to set members(_),
    t(Name,New) notbelongs_to set telephones(_),
    set telephones(_) := (set telephones(_) union {t(Name,New)}).

Other schema in the Z specifications of the ITD system are similarly translated manually into the corresponding ZLOG schema predicates (Appendix A). We do not describe this process in detail, except for the following observations.

- The ∧ and ∨ operations in the schema calculus are the , (and) and ; (or) operators in Prolog.
- The binary relational image operator in Z (and used in schema FindNames) is provided by the rel_image operator in ZLOG.
- The update of the state variable set telephones(_) (in schema RemoveEntry) is achieved by a call to the ZLOG predicate delete_from. This predicate removes the given element from the given set and has the format Element delete_from Set.

3.3 ITD: Executing the ZLOG Program

The above ZLOG program can be interactively executed using the standard Prolog interpreter. This execution can be thought of as prototyping the Z specifications. We first reset the ZLOG system using the built-in ZLOG predicate zlog_start. We assume that there are no elements in the global data sets set members(_) and set telephones(_).

?- zlog_start.
   yes
We first add three persons (from set ‘PERSON’) as members.

?- zcall('DoAddMember'([a],[Rep])). % Make person a a member
Rep = ok
?- zcall('DoAddMember'([b],[Rep])). % Make person b a member
Rep = ok
?- zcall('DoAddMember'([c],[Rep])). % Make person c a member
Rep = ok

Any attempt to add already existing members or members not in set ‘PERSON’ fails.

?- zcall('DoAddMember'([a],[Rep])). % Try to make person a a member
Rep = 'already a member'
?- zcall('DoAddMember'([z],[Rep])). % Try to make person z a member
Type-check failed! Type=set PERSON; Value=z

We now add a few entries (i.e., associate telephones for members).

?- zcall('DoAddEntry'([a,1],[Rep])). % Associate phone 1 with person a
Rep = ok
?- zcall('DoAddEntry'([a,2],[Rep])). % Associate phone 2 with person a
Rep = ok
?- zcall('DoAddEntry'([b,2],[Rep])). % Associate phone 2 with person b
Rep = ok
?- zcall('DoAddEntry'([c,3],[Rep])). % Associate phone 3 with person c
Rep = ok

Any attempt to add already existing entry, entry for non-members or entry involving elements which are not in set ‘PERSON’ or set ‘PHONE’ fails.

?- zcall('DoAddEntry'([a,1],[Rep])). % Try to associate phone 1 with person a
Rep = 'entry already exists'
?- zcall('DoAddEntry'([d,1],[Rep])). % Try to associate phone 1 with person d
Rep = 'not a member'
?- zcall('DoAddEntry'([z,1],[Rep])). % Try to associate phone 1 with person z
Type-check failed! Type=set PERSON; Value=z

It is easy use ZLOG predicates to observe the contents of the current global state.

?- X seteq set members(_). % who are the members?
X = {c,b,a}
?- X seteq set telephones(_). % what are the contents of the directory?
X = {t(c,3), t(b,2), t(a,2), t(a,1)}

Alternatively, we can execute the schema predicates ‘DoFindPhones’ and ‘DoFindNames’ to examine parts of the current global state.

?- zcall('DoFindPhones'([a],[Nums,Rep])). % what phones does a have?
Nums = {2, 1}
Rep = ok
?- zcall('DoFindPhones'([d],[Nums,Rep])). % what phones does d have?
Nums = {}
Rep = 'unknown name'
Schema predicates 'DoRemoveMember' and 'DoRemoveEntry' can be similarly executed and tested.

4. Conclusions

In this paper, we discussed the need to execute and prototype formal specifications so as to test them and to validate them with users. We presented the ZLOG toolkit which contains Prolog predicates to model and symbolically execute the mathematical structures and operators in Z. ZLOG also provides dynamic type-checking and predicates to simulate the schema calculus in Z. We illustrated the use of ZLOG in prototyping a sample Z specification. ZLOG has been used to prototype Z specifications of several large practical systems and was found to be a valuable tool. ZLOG specifications can be incrementally developed, tested, queried, analyzed for properties and executed as prototypes. ZLOG helps to interactively improve problem understanding and detect at the earliest functional shortcomings - a frequent reason causing delayed client acceptance and large overheads in rework. For further work, the theoretical relationships between semantics of Z and the operational semantics of Prolog need to be explored. Also, we are in the process of implementing some refinement operators for Z in ZLOG.

Acknowledgments

I thank Prof. Mathai Joseph, Prof. K.V. Nori and Dr. E.C. Subbarao for their encouragement and support. I also thank all my colleagues in TRDDC for helpful discussions and feedback. Finally, I wish to thank Dr. Manasee G. Palshikar for her vision, patience and confidence.

References

### Appendix A. Some Important Predicates in ZLOG

**Table 1. Some Set Operators in ZLOG**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?A belongs_to +Y</td>
<td>true if the given element A is a member of the given set X; successively</td>
</tr>
<tr>
<td></td>
<td>returns the elements of the given set X if A is un-instantiated</td>
</tr>
<tr>
<td>+A not_belongs_to +Y</td>
<td>true if the given element A is not a member of the given set X</td>
</tr>
<tr>
<td>+X subset +Y</td>
<td>true if the given set X is a subset of Y i.e., X ⊆ Y</td>
</tr>
<tr>
<td>+X strict_subset +Y</td>
<td>true if the given set X is a strict subset of Y i.e., X ⊂ Y</td>
</tr>
<tr>
<td>?X seteq ?Y</td>
<td>true if the given sets X and Y contain the same elements; at least one of</td>
</tr>
<tr>
<td></td>
<td>the two sets X and Y should be instantiated. If Y is un-instantiated, then</td>
</tr>
<tr>
<td></td>
<td>Y is instantiated to the enumerated set containing elements of the given</td>
</tr>
<tr>
<td></td>
<td>set X. Similarly, if X is un-instantiated then Y is returned instantiated.</td>
</tr>
<tr>
<td>+X disjoint +Y</td>
<td>true if the given sets X and Y have no common element</td>
</tr>
<tr>
<td>?N cardinality +X</td>
<td>true if given N is the cardinality of set X; returns the number of elements</td>
</tr>
<tr>
<td></td>
<td>in X if N was un-instantiated</td>
</tr>
<tr>
<td>+X union +Y</td>
<td>set expression which stands for the set X ∪ Y</td>
</tr>
<tr>
<td>+X intersection +Y</td>
<td>set expression which stands for the set X ∩ Y</td>
</tr>
<tr>
<td>+X product +Y</td>
<td>set expression which stands for the set X × Y</td>
</tr>
<tr>
<td>+X difference +Y</td>
<td>set expression which stands for the set X − Y i.e., all elements of X,</td>
</tr>
<tr>
<td></td>
<td>which are not in Y</td>
</tr>
<tr>
<td>+X sym_difference +Y</td>
<td>set expression which stands for the set X + Y (symmetric difference of</td>
</tr>
<tr>
<td></td>
<td>X and Y); X + Y consists of all those elements of X which are not in Y and</td>
</tr>
<tr>
<td></td>
<td>all those elements of Y which are not in X</td>
</tr>
<tr>
<td>empty(+X)</td>
<td>true if the given set X has no elements (i.e., it is empty)</td>
</tr>
<tr>
<td>‘F’ +X</td>
<td>stands for the power-set (i.e., the set of all subsets) of the given set X</td>
</tr>
<tr>
<td>‘FI’ +X</td>
<td>stands for the set of all non-empty subsets of the given set X</td>
</tr>
<tr>
<td>+A add_to +X</td>
<td>adds the given element A to given set A; no action if A is already in X</td>
</tr>
<tr>
<td>+A delete_from +X</td>
<td>removes the given element A from given set X; fails if A is not in X</td>
</tr>
<tr>
<td>delete_set(+X)</td>
<td>remove all elements from the given set X</td>
</tr>
<tr>
<td>+X := +Y</td>
<td>update given set X by the given set Y (i.e., remove all elements from the</td>
</tr>
<tr>
<td></td>
<td>given set X and add all elements of Y to X)</td>
</tr>
</tbody>
</table>
Table 2. Some Relational Operators in ZLOG

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isrel2(+Rel2)</td>
<td>succeeds if given set Rel2 consists only of tuples of arity 2</td>
</tr>
<tr>
<td>dom +Rel2</td>
<td>stands for the domain of the given binary relation Rel2</td>
</tr>
<tr>
<td>ran +Rel2</td>
<td>stands for the domain of the given binary relation Rel2</td>
</tr>
<tr>
<td>+U dom_restrict +Rel2</td>
<td>stands for the set of those tuples t(X,Y) in the given binary relation Rel2 such that X is in the given set U (U ⊆ dom Rel2)</td>
</tr>
<tr>
<td>+U dom_antirestrict +Rel2</td>
<td>stands for the set of those tuples t(X,Y) in the given binary relation Rel2 such that X is not in the given set U (U ⊆ dom Rel2)</td>
</tr>
<tr>
<td>+U ran_restrict +Rel2</td>
<td>stands for the set of those tuples t(X,Y) in the given binary relation Rel2 such that Y is in the given set U (U ⊆ ran Rel2)</td>
</tr>
<tr>
<td>+U ran_antirestrict +Rel2</td>
<td>stands for the set of those tuples t(X,Y) in the given binary relation Rel2 such that Y is not in the given set U (U ⊆ ran Rel2)</td>
</tr>
<tr>
<td>identity(+Rel2)</td>
<td>true if given binary relation Rel2 is an identity relation</td>
</tr>
<tr>
<td>from_to(+Rel2,+DSet,+RSet)</td>
<td>true if dom Rel2 ⊆ DSet and ran Rel2 ⊆ RSet</td>
</tr>
<tr>
<td>rel2_in(+Rel2,+Set)</td>
<td>true if dom Rel2 ⊆ Set and ran Rel2 ⊆ Set</td>
</tr>
<tr>
<td>total(+Rel2,+DSet)</td>
<td>true if for all X in DSet there is at least one tuple t(X, ) in Rel2</td>
</tr>
<tr>
<td>reflexive(+Rel2,+DSet)</td>
<td>true if for all X in set DSet, t(X,X) ∈ Rel2</td>
</tr>
<tr>
<td>irreflexive(+Rel2,+DSet)</td>
<td>true if for all X in set DSet, t(X,X) ⊄ Rel2</td>
</tr>
<tr>
<td>symmetric(+Rel2)</td>
<td>true if for every t(X,Y) in Rel2, t(Y,X) is also in Rel2</td>
</tr>
<tr>
<td>antisymmetric(+Rel2)</td>
<td>true if for every t(X,Y) in Rel2, t(Y,X) is not in Rel2</td>
</tr>
<tr>
<td>transitive(+Rel2)</td>
<td>true if for every t(X,Y) and t(Y,Z) in Rel2, t(X,Z) is in Rel2</td>
</tr>
<tr>
<td>equivalence(+Rel2)</td>
<td>true if given Rel2 is reflexive, symmetric and transitive</td>
</tr>
</tbody>
</table>

Table 3. Some Function Operators in ZLOG

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isfunc(+Func)</td>
<td>true if the given set Func is a binary relation such that for every x there is at most one tuple in Func with x as the first argument</td>
</tr>
<tr>
<td>func_partial(+Func,+DomSet)</td>
<td>true if the given function Func is a truly partial function on DomSet (i.e., there is at least one element x in DomSet such that there is no tuple t(x, ) in Func)</td>
</tr>
<tr>
<td>func_total(+Func,+DomSet)</td>
<td>true if the given function Func is a total function on DomSet (i.e., for every element x in DomSet there is one tuple t(x, ) in Func)</td>
</tr>
<tr>
<td>onto(+Func,+RangeSet)</td>
<td>true if the given function Func is onto RangeSet</td>
</tr>
<tr>
<td>one_to_one(+Func)</td>
<td>true if the given function Func is a one-to-one function</td>
</tr>
<tr>
<td>one_to_one_onto(+Func,+RangeSet)</td>
<td>true if the given function Func is one-to-one and onto</td>
</tr>
<tr>
<td>+Func1 override +Func2</td>
<td>stands for the function obtained by overriding the given function Func1 by the given function Func2</td>
</tr>
</tbody>
</table>

Appendix B. ZLOG Program for the Internal Telephone Directory System

set 'PERSON' ::= {a,b,c,d,e}.
set 'PHONE' ::= {1,2,3,4,5,6}.
set 'REPORT' ::= { ok, 'not a member', 'entry already exists',
                 'unknown name', 'unknown number', 'unknown entry',
                 'already a member' }.

'PhoneDB'([],[],[]) :=
  dom set telephones subset set members.
'AddEntry'( [ 'PhoneDB'([],[]),
[ Name : set 'PERSON', New : set 'PHONE' ], [] ] :-
Name belongs_to set members(_),
t(Name,New) not_belongs_to set telephones(_),
set telephones(_) := (set telephones(_) union {t(Name,New)}).

'Success'([], [], [Rep : set 'REPORT']) :- Rep = ok.

'DoAddEntry'( ['PhoneDB'([],[]),
[ Name : set 'PERSON', New : set 'PHONE' ],
[Rep : set 'REPORT'] ) :-
zcall( 'AddEntry'( [Name, New], [] ) ),
zcall( 'Success'( [], [Rep] ) )
;zcall( 'ErrorNotMember'( [Name], [Rep] )
;zcall( 'ErrorEntryAlreadyExists'( [Name,New], [Rep] )).

'ErrorNotMember'( ['PhoneDB'([],[])],
[ Name : set 'PERSON' ],
[Rep : set 'REPORT'] ) :-
Name not_belongs_to set members(_),
Rep = 'not a member'.

'ErrorEntryAlreadyExists'( ['PhoneDB'([],[])],
[ Name : set 'PERSON', New : set 'PHONE' ],
[ Rep : set 'REPORT' ] ) :-
t(Name,New) belongs_to set telephones(_),
Rep = 'entry already exists'.

'FindPhones'( ['PhoneDB'([],[])],
[ Name : set 'PERSON' ],
[ Nums : 'F' set 'PHONE' ] ) :-
Name belongs_to dom set telephones(_),
Nums seteq {Name} rel_image set telephones(_).

'ErrorUnknownName'( ['PhoneDB'([],[])],
[ Name : set 'PERSON' ],
[ Rep : set 'REPORT' ] ) :-
Name not_belongs_to dom set telephones(_),
Rep = 'unknown name'.

'DoFindPhones'( ['PhoneDB'([],[])],
[ Name : set 'PERSON' ],
zcall( 'FindPhones'( [Name], [Nums] ) ),
zcall( 'Success'( [], [Rep] )
;Nums seteq {},
zcall( 'ErrorUnknownName'( [Name], [Rep] )).

'FindNames'( ['PhoneDB'([],[])],
[ Num : set 'PHONE' ],
[ Names : 'F' set 'PERSON' ] ) :-
Num belongs_to ran set telephones(_),
Names seteq {Num} rel_image rel_inverse set telephones(_).

'ErrorUnknownNumber'( ['PhoneDB'([],[])],
[ Num : set 'PHONE' ],
[ Rep : set 'REPORT' ] ) :-
Num not_belongs_to ran set telephones(_),
Rep = 'unknown number'.

18
'DoFindNames'( ['PhoneDB'([],[])],
[ Num : set 'PHONE',
Names : 'F' set 'PERSON',
Rep : set 'REPORT' ] ) :-
zcall( 'FindNames'([Num],[Names]) ),
zcall( 'Success'([],[Rep]) )
;
Names seteq {},
zcall( 'ErrorUnknownNumber'( [Num],[Rep] )).

'RemoveEntry'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
Old : set 'PHONE',
[] ] ) :-
t(Name,Old) belongs_to set telephones(_),
set telephones(_) := (set telephones(_) difference {t(Name,Old)}).

'ErrorUnknownEntry'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
Old : set 'PHONE',
[] ] ) :-
t(Name,Old) not_belongs_to set telephones(_),
Rep = 'unknown entry'.

'DoRemoveEntry'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
Old : set 'PHONE',
[] ] ) :-
zcall( 'RemoveEntry'([Name,Old],[]) ),
zcall( 'Success'([],[Rep]) )
;
zcall( 'ErrorUnknownEntry'( [Name,Old],[Rep] )).

'RemoveMember'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
[] ] ) :-
Name belongs_to set members(_),
set members(_) := (set members(_) difference {Name})
set telephones(_) := ({Name} dom_antirestrict set telephones(_)).

'DoRemoveMember'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
[] ] ) :-
zcall( 'RemoveMember'([Name],[]) ),
zcall( 'Success'([],[Rep]) )
;
zcall( 'ErrorNotMember'([Name],[Rep]) ).

'AddMember'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
[] ] ) :-
Name not_belongs_to set members(_),
set members(_) := set members(_) union {Name}.

'ErrorAlreadyMember'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
[] ] ) :-
Name belongs_to set members(_),
Rep = 'already a member'.

'DoAddMember'( ['PhoneDB'([],[])],
[ Name : set 'PERSON',
[] ] ) :-
zcall( 'AddMember'([Name],[]) ),
zcall( 'Success'([],[Rep]) )
;
zcall( 'ErrorAlreadyMember'([Name],[Rep]) ).