RAPID PROTOTYPING OF SOFTWARE SPECIFICATIONS IN Z

By

WU CHUN PONG

A Thesis

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Information Engineering

The Chinese University of Hong Hong

December 1993

Acknowledgement

I would like to express my sincere gratitude towards my supervisor, Dr. Edmund M. K. Lai for his invaluable suggestions in this research and comments in preparing this thesis.

Abstract

Current software tools reported in the literature for animating formal specifications written in the Z notation are incapable of handling data declared over infinite sets. In this thesis, we have shown that this capability is often essential to specifications that involve numbers. A feasible solution is to use concepts in delayed evaluation and constraint satisfaction to handle infinite numerical sets. A system called ZCLP(R) has been developed and the implementation is based on the constraint logic programming language, CLP(R). It consists of an editor and two translators. One translator converts Z to CLP(R) and the other converts Z to LaTeX format. The former is for execution while the latter uses a software called "fuzz" for printing and type checking. ZCLP(R) is also capable of handling concepts of bag and object orientation. Some examples are presented to illustrate the effectiveness of the ZCLP(R). Finally, specifications writing experience is also given.

Contents

Chapter 1

Introduction

1.1 Formal Specification Methods

Formal specification methods arise as a result of combining mathematics and computer science. They offer the main advantage that a specification, once it has been written, uniquely describes the behaviour of the system to be designed. Formal methods are able to provide a clear and an abstract description of a system. This merit is contributed by the preciseness of the underlying mathematical language. In other words, ambiguities in natural languages will not be found in formal methods.

A user can specify his ideas without concerning the implementation details. Hence, it facilitates the break down of system design into small blocks. Each block has its own specifications. These specifications can be transferred among people without misunderstanding. On the other hand, people will find it easy to modify the specifications if there is any update or modification.

In addition, mathematics can help in proving the consistency among specifications. Reference [35] chapter 1 has shown a good example. It implies we can reduce errors in the early abstract design stage. By contrast, any changes after the implementation will demand great cost and manpower. This is important for any critical systems where no hidden bugs are tolerated. For example, we have the train information system described in [30] and the project of software and hardware specifications used by the US Department of Defense [27].

With the above advantages, formal methods have been gaining acceptance in both software and hardware design communities [4, 5, 27]. But insufficient tools and education are still a major hindrance. This is particularly true for the newly developed formal specification notation Z.

1.2 The Z notation

The Z notation [35] is a formal specification method based on set theory. It was initially designed for specifications of real large-scale software systems and had been successfully applied to a number of industrial software development projects [12]. Z makes use of set theory notations to form schemas – blocks each defining a state or an operation. A schema typically consists of three parts: title, declaration and predicates. A group of schemas defines an abstract model which specifies the system design.

Figure 1.1 illustrates the design flow based on formal methods. Starting with the user requirements, an initial specification is written. This specification is then verified against user requirements to ensure that there is no discrepancy between the two. A common verification method employed is called *animation*. This means executing the specification on a computer. Since a typical formal specification language, including Z, is not executable, it must first be translated into an executable computer language. The aim of animation is not to provide an exact or detailed representation of the system specified, but rather to illustrate the behaviour of the system based on an execution model.

Some animation tools for Z had been developed and reported in the literature [22, 39, 7]. Among them, *SuZan* and *EZ* seem to be more complete. Both of them translate Z schema into Prolog clauses for execution, where the clauses are linked with a pre-defined Prolog library. A major shortcoming of both tools is that they can only handle variables declared over *finite* data sets. Infinite data sets, however, occur in many practical applications, most notably with numbers. They include the sets of natural numbers and integers. Any finite interval over the set of real numbers also consists of infinite number of real numbers if the precision is required to be unlimited.

1.3 Overview of Thesis

In this thesis, we present our Z animation system ZCLP(R) that is capable of handling the *infinite* sets of integers and real numbers. It consists of an editor

and two translators. The translator of Z to CLP(R) is an automatic process with a simple user interface for test data input. Our approach is to use concepts in delayed evaluation and constraint satisfaction to handle infinite numerical sets. The implementation is based on the constraint logic programming language, CLP(R) [20]. Another automatic translator converts Z to LaTeX [25] format using a software called "fuzz" [36] for printing and type checking. ZCLP(R) is also capable of handling concepts of bag and object orientation. In the following chapters, we shall first give a brief review of Z, Prolog, CLP(R), *SuZan* and *EZ*, followed by detailed descriptions of our animation system as well as discussions on how infinite number sets are handled. Finally, the effectiveness of ZCLP(R) is illustrated by examples.

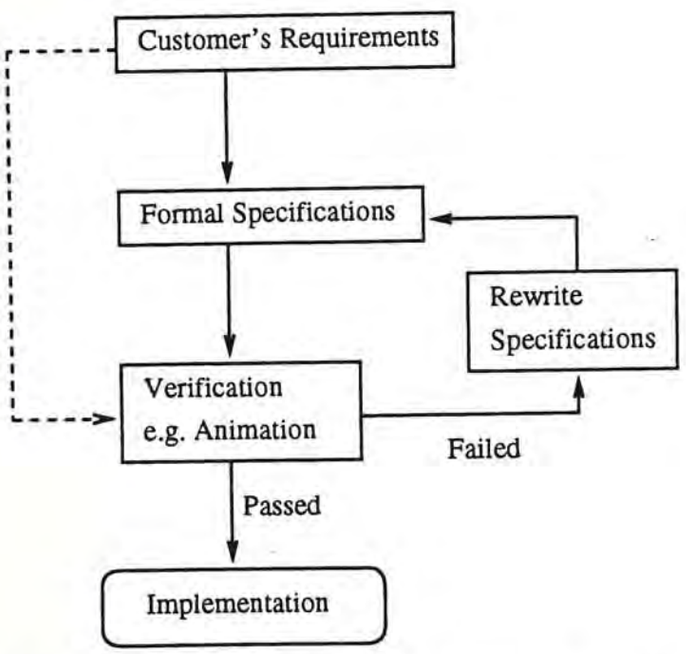


Figure 1.1: Design Flow Using Formal Methods

Chapter 2

The Specification Language Z

2.1 Background

Integrating mathematical techniques in software engineering is the characteristic of formal methods. Z is one of the instances which is based on Zermelo-Fraenkel set theory. It was initiated by Jean-Raymond Abrial in France and developed by the Programming Research Group of the Oxford University in England. Z has been evolved over a decade and still under researched.

The growth of Z has been proved in the various applications of software design. Examples can be found in [12, 21, 28, 15, 33, 34] which cover the area of system design, communication protocols, transaction processing, behaviour modelling and so on. Its usage can range from high level of abstract ideas, to intermediate or low level of implementations.

2.2 Structure and Characteristics

The Z's structure is mainly of schema boxes. A schema is a group of statements bounded by lines, where the appearance enhances module visualization. A typical schema consists of schema title, declaration and predicates.

Title

Declarations

Predicates

The "Title" is the schema's title itself. The "Declarations" is the place where variables of different types, schema calculus are defined(schema calculus is the operations on another schema and will be discussed later). The relation among the variables and their properties are founded in the "Predicates" part.

We also have the axiomatic and generic schemas. The former describes global variables and their constraints while the later defines some common functions upon objects of different types.

Axiomatic description

Declarations

Predicates

Example

$counter : \mathbb{N}$

$counter < 1000$

Generic definition

$$[Params]$$
$$Declarations$$
$$\vdash$$
$$Predicates$$

or

$$Title[Params]$$
$$Declarations$$
$$\vdash$$
$$Predicates$$

Example

$$[X]$$
$$head : \mathrm{seq}_1\, X \longrightarrow X$$
$$\vdash$$
$$\forall s : \mathrm{seq}_1\, X \bullet head\ s = s(1)$$

As found from the above example, the contents of the schema are constructed by set notations. These notations have been well defined in the set theory. The statement in the predicate part can be read as "for all s in the finite sequence of type X such that head s is the first element of the sequence s".

A set is a group of elements which share a common property. This property can be regarded as a type in Z. For example, we have the type natural number. Variables can be declared to be within this type. Among them, we have plenty

of notations to work out their relationships. These notations can be classified as logics, sets operators, relations, functions, sequences and bags. They all belong to the area of discrete mathematics. The book Z notation [35] has provided a well reference to their definitions.

Some notations are special and form the main feature of Z. They are the decorations. Variables can be specified as an input or an output property by appending characters "?" and "!" respectively. A prime can also be used to identify a new state of the variable after some operations. These decorations has extended to the whole schema structure. Symbols Δ *or* Ξ of a schema represent a changing state or not. Actually, $\Delta schema$ is equivalent to $schema \wedge schema'$, a calling of the schema with states before and after operations. Then, all the variables in $schema'$ are also decorated by prime. These notations have favoured Z as a state or property oriented.

$\Delta Schema$

$Schema$

$Schema'$

$\Xi Schema$

$\Delta Schema$

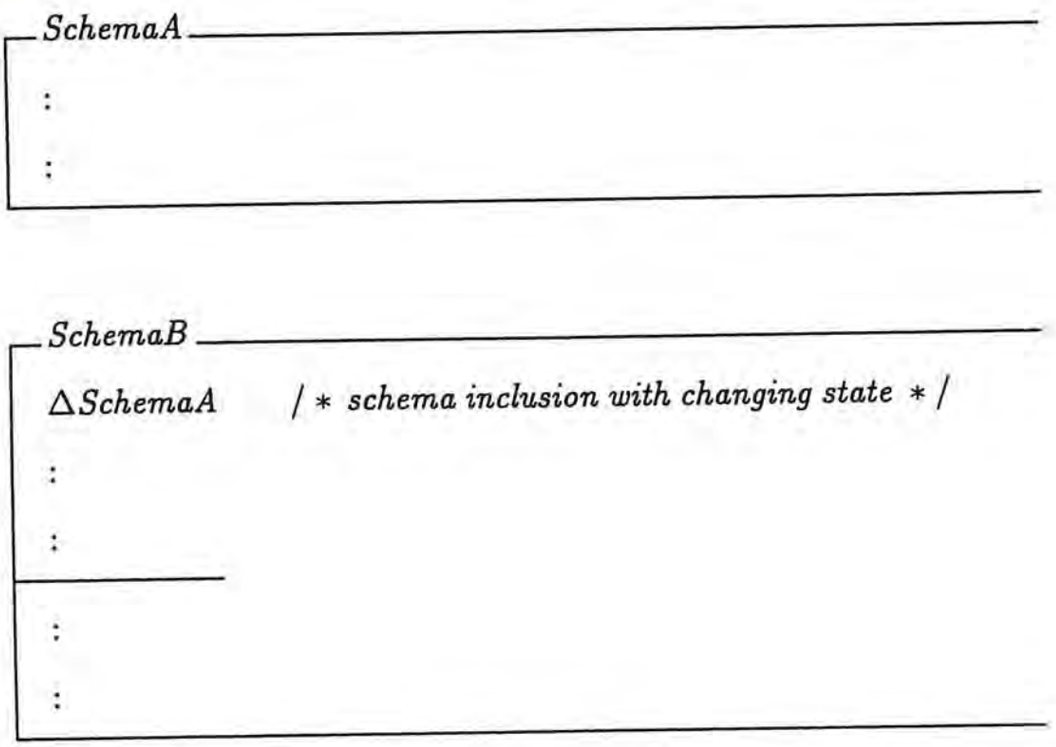
$NoChange$

In fact, the schema decoration belongs to the class of schema calculus.

Schema calculus defines various operators which relate schemas in a manageable way. It consists of the following topics:

- Schema inclusion
- Schema decoration
- Schema disjunction
- Schema conjunction
- Schema negation
- Schema hiding operators
- Schema composition
- Schema preconditions

The explanations of each item can be found on [29]. Of which, schema inclusion, decoration, disjunction and conjunction are used frequently. Schema disjunction and conjunction are the logical "and" and "or" between schemas. For example, $SchemaC = SchemaA \wedge SchemaB$. Schema inclusion brings another schema into the declaration part of the present schema. In effect, the variables and constraints of the calling schema are included implicitly. Together with decoration, these properties will help in system design of building blocks of functions.



2.3 Object Orientation in Z

Modern software programming is always talking about object orientation. Unquestionably, object orientation in software engineering can benefit structuring, understandability, modifying and future development. Conventional Z writing, however, fails to specify classes of objects. A class is a collection of objects which share common characteristics. Class can be built on another class with bringing along the previous properties. The technique is called inheritance. It allows classes can be incremented in a proper way. Many researches have been conducted in extending Z to this area. All of them are trying to work out a good representation. A survey has shown in literature [37] that there are three methods of Z object orientation.

2.3.1 Hall's style

Hall [9] describes a writing style on standard Z for the object orientation. It has been implemented in a project called CASE [2]. Within this approach, an object has a self identity which is never changed.

Object

$$self : IDENTITY$$

:

:

Then, a system state may be a collection of the objects. No objects in the system are duplicated by means of different entities. This can be ensured by a function relating identity and object:

System

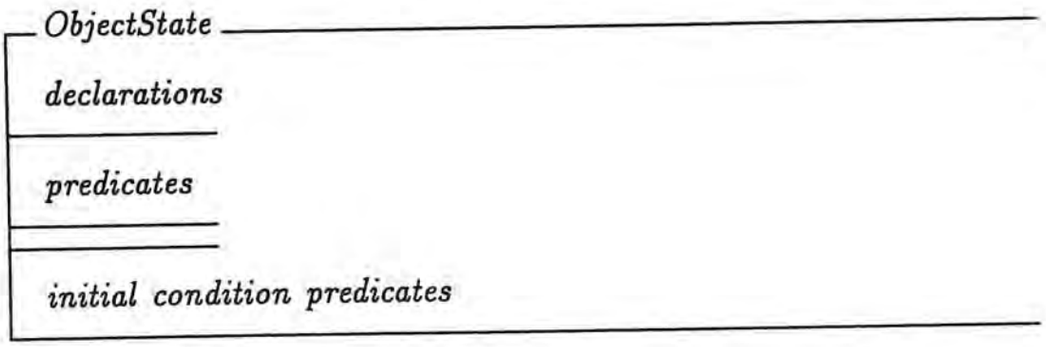
$$objects : \mathbb{P}\ Object$$

$$idObject : IDENTITY \pfun Object$$

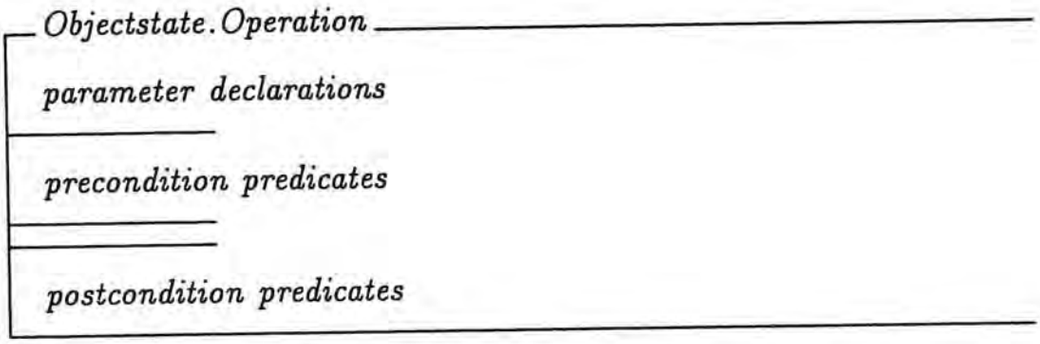
$$idObject = o : objects \bullet o.self \mapsto o$$

2.3.2 Schuman and Pitt's variant

A object state schema can be described by the structure:



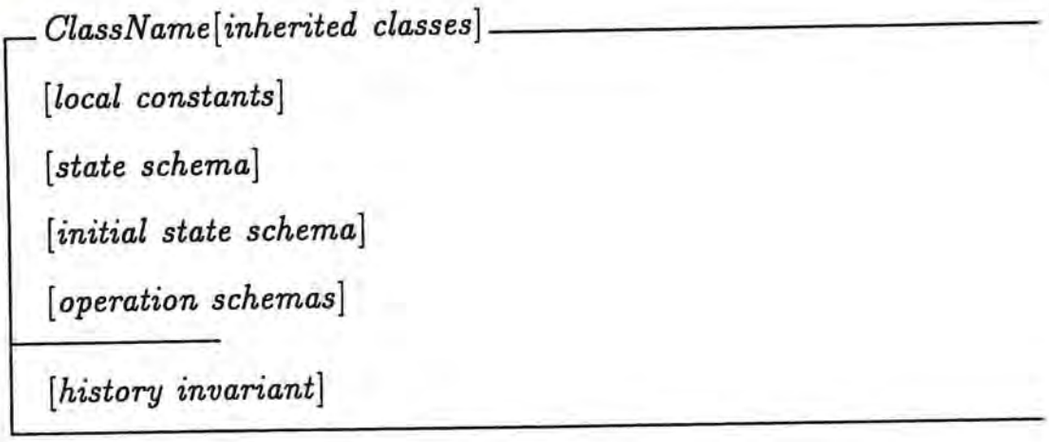
The operation schema on that object is titled by adding a dot extension. It also has three components.



Similar to other object orientation methods, variables and constraints in the object state have passed implicitly into the operation schema.

2.3.3 Object-Z

In this approach, the outlook is an enhancement of module visualization. An object class schema can contain sub-schemas for its state definitions. The development and usage can be found on reference [6, 32]. The formal structure is of the form:



2.4 Execution in Z

Another research area of extending Z is to develop tools for execution. A debate concerning whether execute specifications or not has been found in literature [8] and [10]. In summary, Hayes and Jones prefer not to execute the specifications with the following main reasons:

- Executions with some data sets do not represent the general case of specifications. In other words, a good testing does not guarantee the absence of bugs.
- Executable language concerns the implementation details and reduces the abstract level of specifications.
- Specifications can involve equations of inverse operations. In general, it can not be executed directly. For example, we can define most integer square root R of N in terms of square, $R^2 <= N < (R+1)^2$. It is of course an indirect implication.

- Combining clauses, negation of clauses and using quantifiers $\forall$ $\exists$ may result in an infinite set.

- It is easier to implement an efficient system regarded to an abstract specification than to follow an executable specification. The later may hinder the structure or the data set of the system.

Three years later, Fuchs presented a paper to object the view. He made use of logic specification language(LSL), a language of Prolog's extension, to support his arguments.

- Correctness is more important than the expressive power of a specification. Executing of specification can provide a conceptual and behavioural model. Then, the early feedback can reduce the discrepancies between informal ideas and formal languages as well as the costs of implementation.

- The LSL, a declarative language with Horn Clause logic, is suitable for specification language. It is expressive and executive. It has shown that non-computable clauses can be expressed in LSL with slightly decreased in abstract level, a level of still not concerning the implementation details.

- Specifying by inverse can be done with the generate-and-test technique. For the example of finding the square root, a recursive generator of natural number can be defined such as $natural\_number(R)$. Values of R will be generated and tested with the constraints.

- The infinite set can be bounded by adding a limit into the generator. For example, we can define $limit\_natural\_number(R, Limit)$.

- The executable specifications can form the basis of the implementation provided they can describe the system completely and abstractedly.

In fact, if the abstract level difference between non-executable and executable specifications is small, it is valuable to further investigate the feasibility of execution. At least, a quick and concrete feedback will enrich the confidence of the specifier of what he has written.

The Fuchs' paper has given a great hint in the way of execution. It is the use of Prolog extension. One of our finding in this area is the software "Constraint Logic Programming CLP(R)". With CLP(R), the infinite set of numbers can be solved without any generators or limit values. We will discuss the research in the following chapters.

2.5 Animation of Z Specifications

2.5.1 Prolog

Prolog, programming in logic, is an executable language using logic to describe the relationship between condition and conclusion. The programs are so simple and clear that large amount of equivalent low level programming would be required. With this expressive power, Prolog has been applied successfully in the field of Artificial Intelligence.

The expressive power of Prolog is mainly due to the use of the Horn clauses.

Reference [26] has provided a good picture of Horn clauses in logic programming, figure 2.1.

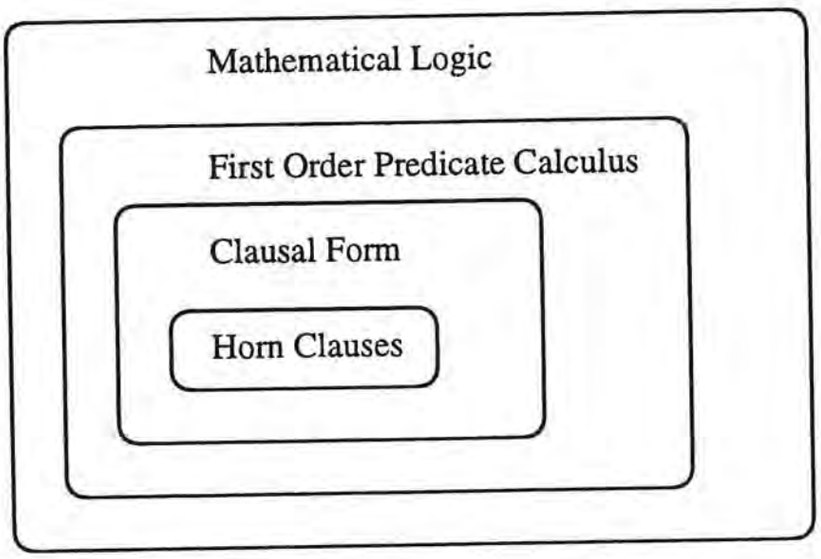


Figure 2.1: Layers of logic programming, depicted from [26] p.86.

Under the set of mathematical logic, first order predicate consists of "Terms" and "Atomic formula". Variables and constants, $X_1, X_2, ..., X_n$, are called terms. A function f with arguments of terms, $f(X_1, X_2, ...X_n)$, is also called a term. Predicate P with terms' arguments, $P(X_1, X_2, ..., X_n)$, forms the basic atomic formula. Then, the first order calculus relates atomic formulas by negation, conjunction, disjunction, implication, and equivalence with corresponding symbols $\neg \quad \wedge \vee \leftarrow \quad \leftrightarrow$. The calculus also includes areas of existential and universal quantifiers, $\exists \ \forall$.

After building up first order logic, we can define a Clausal Form of:

$$B_1, B_2, ..., B_n \leftarrow A_1, A_2, ..., A_n$$

where B_n and A_n are the atomic formulas. We can read the clause as conclusions $B_1, B_2, ..., B_n$ hold if conditions $A_1, A_2, ..., A_n$ are true. If only at most one conclusion is found in the clause, say B or nothing, it is named the Horn clause.

In Prolog, the head of the clause or conclusion, can be stated by a fact. For instance, "Mary is elder than peter" can be written as $elder(mary, peter)$. Or, the head is a part of a rule likes $elder(X, Z) : -elder(X, Y) \wedge elder(Y, Z)$. The symbol ": −" represent the function "if" where the left hand side conclusion holds if the right hand side condition is true. It is noted that an upper case of the first character of a word stands for a variable, while the lower case means a constant, a fact or a functor. In addition with another fact $elder(john, mary)$, we can query the system whether $elder(john, peter)$ is true or not. The answer is of course "yes".

Another characteristic of Prolog is its recursive power. Let us consider the predicate:

```
member(X, [X|_]).
member(X, [_|Y]):- member(X, Y).
```

The square brackets represent a list. Symbol "|" separate the first and the remaining elements. Anonymous variable "_" does not care any actual value. Then, we can ask the question of $member(e, [a, b, c, d, e, f])$. Prolog will first

attempt the first line, e.g. $member(e, [a \mid \_])$. If it fails, second line will be executed. But it is actually a retry of the first line with the remaining elements of the list. This recursive trial will terminate if a match element is found or the end of list is reached.

On inverse, we can query $member(X, [a, b, c, d, e, f])$. Prolog will backtrack the search tree and generate answers $X = a, b, c, d, e, f$ individually upon request. In fact, the searching technique in the rulebase is the depth-first-search. Details explanation and usage of Prolog can be found on [38, 1].

2.5.2 Translation Z into Prolog

Up to now, no compiler or executor exists for Z language. Prolog predicates, however, can be directly executed. This is the reason of trying to link both languages together. Translating Z into Prolog, is an indirect means of executing Z. This translation is possible because both languages are based on first order logic.

The first order logic is so simple that we can perform a one-to-one mapping from Z into Prolog. Logics "$\neg \quad \wedge \vee$" can be mapped to "*not* , ; " in Prolog. Set operators such as $x \in S$ can be interpreted as $member(X, S)$. Similarly, relation $x =$ dom R and function $X \pfun Y$ can be constructed as $dom(R, X)$ or $partial\_function(X, Y)$. This translation is direct and simple.

Of course, Prolog does not recognize the functors *dom* and *partial_function*,

or even the *member*. As in Z, their meanings are pre-defined. Z specifier has already known their definitions but it is not true for the Prolog system. Therefore, a Prolog library is needed to store the predicates of these functions. A sample from the last section has constructed the definition of *member*. As a result, the library can be recalled during execution.

Executing Z by Prolog, however, no formal proof has been given for this translation. We can challenge its correctness. But it is not our main goal. The goal is to animate or execute Z so that some rapid feedback can be obtained. A correct and executable Prolog predicate has reflected a good Z specification to a certain degree.

2.5.3 Related Works

Two approaches to animate Z specifications have been suggested by West and Eaglestone [39]. They are, namely, "formal program synthesis" and "structure simulation". The former approach converts the higher order theory of Z into first order logic and then translates it into Prolog. West and Eaglestone have shown that "formal program synthesis" requires human assistance in the translation process and thus cannot be fully automated. On the other hand, "structure simulation" maintains a similar structure to that of Z. It is the idea described in the previous section. With this approach, Z schema statements are translated into Prolog predicates directly based on a pre-defined library of set operations. It is possible to implement an automatic one-to-one translation from Z to Prolog based on this model. This model has also been found in two animation

researches, *SuZan* [22] and *EZ* [7].

SuZan contains a well defined library called "Mathias" [23]. This library not only works for Z translation, but also forms the basis of a Prolog animation tool for discrete mathematics which covers a number of areas including set, logic, vectors and matrices. Together with the library, the translation of Z into Prolog predicates is semi-automatic. Afterwards, reordering of predicates is sometimes needed to prevent unnecessary backtracks in order to improve the efficiency. Unfolding some library's predicates may also be required to remove duplications.

EZ is an automatic translation tool. The method is to use a look-up list predicate to instantiate variables from a given finite data set. After translation, a control system must be added to animate the model by forward chaining or backward chaining. It is the predicates telling the sequence of program running. This control system takes the advantage of Prolog's backtrack technique so that alternate paths can be found.

Unfortunately, both *SuZan* and *EZ* can only handle variables declared over finite data sets. The reason is that variables' values are generated and tested with constraints. It is not practical for the infinite set such as numbers. For instance, a schema describing the relationship among the edges' length of a right-angle triangle: [24]

PythagTraids

$x!, y!, h! : \mathbb{N}$

$(x * x) + (y * y) = (h * h)$

will be translated into the following Prolog predicates:

```
posnum(1).
posnum(N):- posnum(N1), N is N1+1.


pythagtri(x,y,h):-
        posnum(x), posnum(y), posnum(h),
        0 is h*h - x*x - y*y.
```

If the predicate *pythagtri* is executed with $x = y = 1$, the program will run into an infinite loop of fail and backtrack. The system tests a value for h which is incremented by one at each time. It will be a serious problem if the schemas contain extensive arithmetic operations.

Chapter 3

Incorporating Real Numbers in $\mathbb{Z}$

Starting from the primary school, we have learned that the number system consists of natural numbers $\mathbb{N}$, integers $\mathbb{Z}$ and real numbers. Real numbers is the largest set. It includes rational and irrational numbers. The irrational numbers is a mysterious subject that many people had tried to work out its definition as well as the whole set of real numbers in the past. Not until the end of nineteenth century, Richard Dedekind and George Cantor had given an abstract definition in terms of rational numbers. Their full explanations can be found in [3, 31]. Afterwards, the rapid development in real numbers such as limit and continuity have proven real number is an essential concept and tool in the twentieth century.

3.1 Dedekind Cut

A Dedekind cut is a subset α of the set of rational numbers R. One characteristic of set α is that it does not have a minimum. For instance, we can define a cut based on rational number r_0 as:

$$\alpha = \{r : r \in R \text{ and } r > r_0\}. \qquad (*)$$

Omitting the mathematical proof, we cannot find a minimum rational number tending to r_0. Once we locate a value r_a, we can find another value r_b such that $r_a > r_b > r_0$. On the other hand, the set complement of the cut $\alpha^c = R - \alpha$ has a maximum value equal to r_0. If we cannot determine the maximum value, or α is not in the form of (*), it is an irrational cut.

The collection of cuts, rational and irrational, forms the basis of real numbers. Together with the addition, multiplication, and order over the set of cuts, the real number system is well defined.

3.2 Cantor's definition

Cantor's definition starts from the Cauchy sequence. A Cauchy sequence $\{a_n\}$ is a sequence satisfying the condition:

$$\exists\, n, m : \mathbb{Z};\ N, e : R \mid n, m > N \bullet \mid a_n - a_m \mid < e.$$

It is not necessary that the sequence converges. For example, the irrational number $\sqrt{2}$ can be represented by a Cauchy sequence:

$$\{1.4,\ 1.41,\ 1.414,\ 1.4142,\ 1.41421,\ 1.414214,\ \ldots.\ \}$$

If there is another sequence $\{b_n\}$ such that $\{a_n - b_n\}$ converges to 0, i.e.

$$\mid a_n - b_n \mid < e.$$

for $e > 0$ and $n > N$, we define the two sequences are equivalent and are in the same class. In other words, each Cauchy sequence $\{a_n\}$ is unique to an equivalence class. Similar to Dedekind cut, addition, multiplication and order are defined on these classes. The collection of the equivalent classes is the real number system.

3.3 Practical approach

The classical definitions of real numbers by Dedekind cut and Cantor use the abstract level of set theory. The approaches are indirect because they are in terms of set and sequence. In fact, they are not practical in computing programs. All we concern is the method of computing and incorporating in Z.

A real number can be of infinite decimal length. It is meaningless and impossible to perform such an endless calculation. Instead, we can specify a precision value or a fixed decimal point. Any decimal numbers can be cut or added zeros to achieve the decimal point and participate in the arithmetics. But this precision value should be machine independent. A problem arises because we do not know all machines' floating point power. Moreover, it will limit our choice

of decimal length.

Again, a possible solution is to use Prolog. We can interpret a real number as a list. Each integer, minus sign or decimal point is an element of the list. Addition, subtraction, multiplication and division can then be calculated in the way of human's thinking. Calculated result of a pair of elements is passed to the neighbour element. It is a basic method that places no limit on the precision level. So, a system of Prolog predicates called *Real_math* has been constructed. It can simulate the human's calculating procedures. For example, if we want to have a precision of ten decimal digits and perform the calculation 123.123456789 / 123.456, we can input:

divide([1,2,3,.,1,2,3,4,5,6,7,8,9,0], [1,2,3,.,4,5,6], Quotient, Remainder).

The first argument in *divide* is added a zero to determine the accuracy. The system will reply:

Remainder = [., 0, 0, 0, 0, 1, 1, 2, 2, 7, 2] Quotient = [1, ., 0, 0, 0, 0, 0, 6, 3]

Incorporating real numbers into Z can be as simple as just writing down the type *Real* in schema's declaration part. From the above, mathematical operators "+ - * /" can be translated into corresponding predicates. Unfortunately, we can not do it in this way if we choose the software CLP(R) to be our target tool. It is because its delay mechanism is not applied to list. This property will be explained in the next section. After consideration, employing CLP(R) in solving infinite numerical set is more important than the precision problem.

Chapter 4

Constraint Logic Programming and CLP(R)

4.1 Constraint Logic Programming

In the area of logic programming, many researches have been conducted in Prolog's extension. This extension, however, may stray from the semantic properties of logic programs. Thus, it motivates Jaffar, Lassez and Maher [17] to propose a "logic programming scheme". The scheme aims at using logic programming to reason constraints. Alternatively, the scheme is called "Constraint Logic Programming CLP". Unlike other extensions, the scheme first defines a class of languages for forming a formal framework. This is achieved by using the concept of the definite clauses. Also, the framework has the semantic unification where two terms which are not syntactically identical can be considered equal. Consequently, Jaffar and Lassez [18] showed that the framework defined was more general than the one of logic programming.

Overall, literature [18, 19, 20] have shown that CLP languages contain the following characteristics:

- They are soundly based.
- They have great expressive power.
- An efficient implementation can be constructed.

Under CLP scheme, the domain of computation remains unspecified. We can have CLP(X) where X is the domain of the language applied. Prolog, Prolog II and Prolog III can be considered as the class' instances. Prolog II is a Prolog's extension which can solve equations over infinite trees. And, Prolog III expands the domain including areas such as linear arithmetics, booleans and strings. Similarly, Jaffar, Michaylov, Stuckey and Yap [20] developed a system called CLP(R). Unlike Prolog II, the constraint solver in CLP(R) works implicitly to the users.

4.2 CLP(R)

Constraint is a problem which can be solved by:

- Imperative method. It is the simplest way where constraint is treated as tests or assignments. If variables are grounded, they are tested by the constraint. For instance, test(X):- X > 3 is executed only at X is given. If assignment " = " is used, variable of one side can be determined provided the other side is grounded. Or, equality is checked for both given sides.

- Local propagation. The operation involves a delay mechanism but applies to local predicate. Constraints are solved if some variables can be determined with sufficient conditions. Then, a local propagation occurs to pass the newly grounded variables to other constraints. In this method, the order of constraints is important. A drawback is that constraints may fall into cyclic dependency [13, 20].

- Constraint solving method. This method treats constraints declaratively while the above two fail. The declarative nature can free programmers from considering the order of constraint collection. For example, we can write:

  test1(X):- X > 0.
  test2(X):- X < 1.
  test3(X):- X > 5.

  and then query the combined conditions:

  ?- test1(X), test2(X), test3(X).

  the answer is of course "No".

  An instance of this class is CLP(R). Its solver can determine the solvability of linear constraints. Non-linear constraints are treated by the local propagation method.

In CLP(R), the R represents constraints over real arithmetic terms of uninterpreted functors. Its solver involves a delay and wake-up mechanism where

non-linear constraints are delayed until the required variables are grounded to make the constraints linear. More insight, they are filtered by an inference engine, an engine/solver interface, an equation solver and an inequality solver. The strategy is that different constraints are tackled by different algorithms so that the solver can fulfill the requirements of:

- having a good average behaviour.
- adding new constraints would not resolve the previous constraints again.

4.3 Example of CLP(R)

A care in CLP(R) programming is that it has type difference instead of free type in Prolog. If the predicates involve mathematical notations such as ">= < *sin*" , the related variables would be identified as real numbers. The real variables will be delayed if they are not determined yet.

For example, $Z = cos(X)$ will be delayed until X is grounded.

Or, the predicate

ele(V, I, R, P):-
V = I*R,
P = V*I.

will be delayed until any two variables are grounded by other constraints. So, we can ask

?- ele(3, Current, Resistance, 4.5).

and the system will return :

Resistance = 2

Current = 1.5 .

Even the constraint is not linear, required variables are not yet determined, CLP(R) will try to simplify the relation.

For example,

?- ele(3, Current, Resistance, Power).

will return :

Current = 0.333333*Power

3 = Current * Resistance .

Although it is an unsolved answer, the simplified equations themselves describe a correct response. For more examples, please refer to [14].

Chapter 5

The ZCLP(R) Animation System

5.1 Design Philosophy

The purpose of the system is to execute Z through an indirect means by translating Z schemas into predicates of programming logic (Prolog). Under Prolog system, the execution of predicates can act as a validation to Z. This possibility has been discussed in section 3.2.

In the design of our animation system, we considered three important aspects:

A) The system should incorporate real numbers in Z. It is a necessary and common type definition. The problem of involving this type in the translation of Z into Prolog has been discussed in literature [24] and section 3.3, where the

infinite set and precision of real numbers are the key points. The generate-and-test of numbers cannot be used in standard Prolog because it may result in an endless loop. Therefore, an extension to standard Prolog is needed.

One of the solutions is by using constraint logic programming. Actually, this method is under the scheme of CLP system developed by [18]. Prolog II, Prolog III and CLP(R) are the instances. Particularly, CLP(R) is designed for real number calculations. Its constraint solving algorithm uses a delay mechanism where variables of real numbers are not evaluated until there are sufficient conditions. It is an open software obtainable in the unix network public domain. On summary, CLP(R) has the properties of:

- Constraints can be declaratively expressed.
- Non-linear constraints will be delayed and then solved when it becomes linear.

As a result, system of equations can be solved or simplified by CLP(R). We decided to use CLP(R) for the animation system because of its ability to handle real numbers and its availability.

B) It is worth introducing object oriented Z into the system. Literature [32, 6, 37] have shown that object class can benefit structuring, understandability, updating and modifying in software engineering. Various methods of this extension in Z have been discussed in section 2.3. Among them, the Hall's style is the least and simplest changes to the conventional Z writing. Similar to this style, a schema can be treated as a property or class to be recalled in others'

type declaration. A single schema box format is still preserved. For example,

SchemaA

$A1 : TypeA1$

$A2 : TypeA2$

$A1 >= 10 \quad /* \textit{ constraint } A */$

$A2 >= 20$

SchemaB

$B1 : SchemaA$

$B1.A1 > 30 \quad /* \textit{ constraint } B */$

$B2.A2 = B1.A1 - 40$

Clearly, a variable in SchemaB has sub-variables of types defined in SchemaA and is represented by dot extension, namely B1.A1 and B1.A2 . These variables are constrained by SchemaB as well as SchemaA. The later is a hidden implication followed by the class definition. It is useful if there are several objects belonging to a common class so that the class's properties need not be redefined. Then, the approach would result in a well organized tree structure as shown in figure 5.1.

C) An automatic translation system is required. It should not be necessary for the specification writer to take care of the translation process or even the Prolog language.

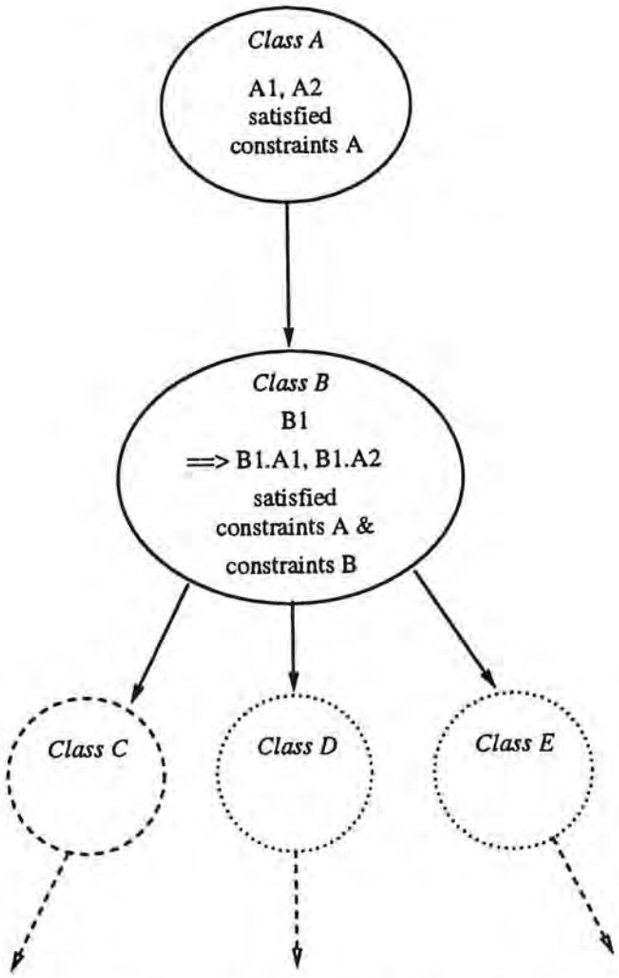


Figure 5.1: Tree structure of object classes

5.2 Implementation Strategy

Since Z is still not standardized, it lacks supporting tools. Normally, most researchers built up their tools themselves. A useful software called *Fuzz* is provided by J.M. Spivey [36]. It can check Z specifications for the compliance of Z scope and type rules. Schemas are written in LaTeX format and fonts of Z notations are available for printing.

The specification process starts with editing and thus an Z editor is essential.

After editing, the file created can be used for interpretation. Two translators are needed, one is the Fuzz translator for printing, type checking and one is Prolog translator for executing. The model of Prolog translator will be similar to the ones developed in [24, 7, 39] and have been discussed in chapter 3. They have shown that the generate-and-test model is applicable to the animation of some simple schemas. With this model, a Prolog's library is needed for executing set operations.

The whole picture can be expressed by figure 5.2.

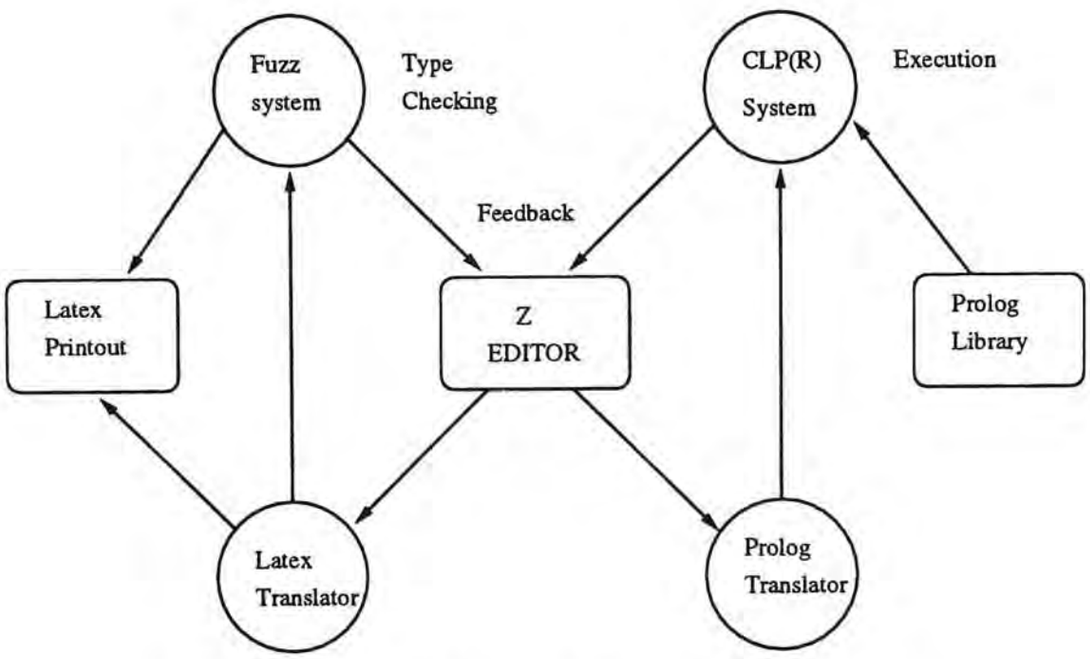


Figure 5.2: The design of the system ZCLP(R)

5.3 Z editor (ZEDIT)

As [16] points out that complex structure editors may restrict users the degree of freedom, the design of editor is as simple as a text editor shown in figure 5.3. The editor is written in C language of graphics mode. Special Z schema's lines and symbols are available on screen. Once the schema key is pressed, the editor will enter the schema mode. Schema lines will be drawn automatically until the end of schema box is input. The final work is saved as an ASCII file. In the file, numbered markers are represented for those special Z notations.

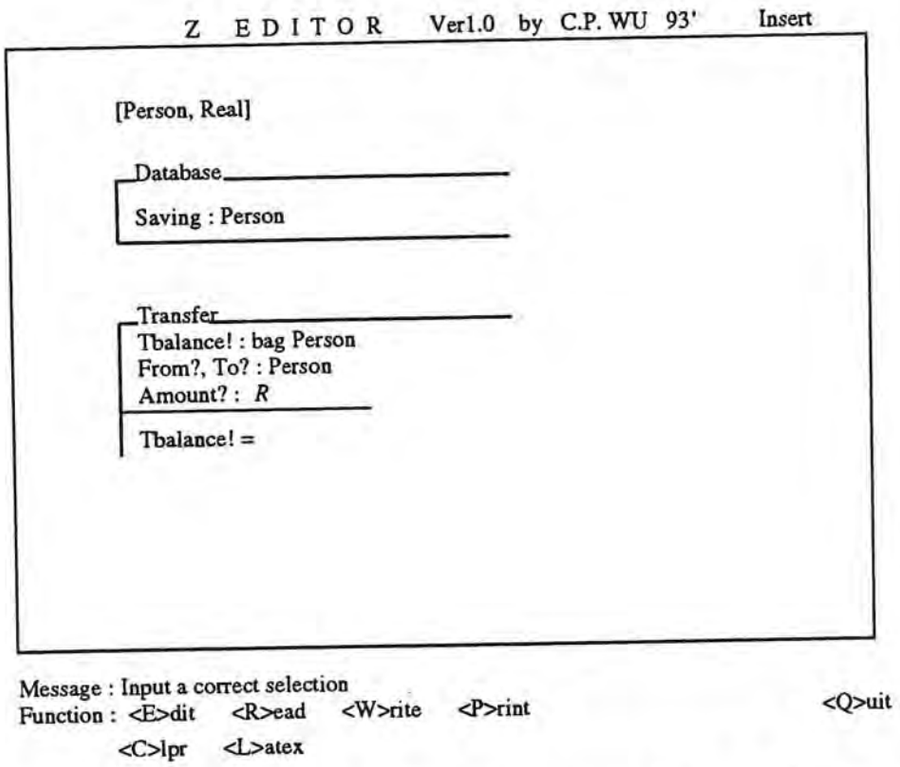


Figure 5.3: A view of the main screen of the Z Editor

5.4 Prolog Library for set operation (ZCLIB)

The task of the library is to release the burden of translation. Some common set operations can be recalled from this library such as *union* and *intersection*. Their implementations are referred to the definitions described in [35].

5.4.1 Basic needs for the Library

Most functions in the library will work on list operation. However, the built-in functions of CLP(R) are very limited. We have to define them by ourselves which are:

1. member(X, list) – membership of element X in the list.
2. append(ListA, ListB, ListAB) – append one list to another.
3. delete(X, List, Result) – delete element X in list.
4. length_list(List, N) – the length N of a list.
5. findall(X, Pred, Result) – find out all elements X satisfied with the constraint "Pred" and then form the list "Result".
6. setof(X, Pred, Result) – similar to findall predicate but remove any duplication of the list "Result".
7. reverse(ListA, Alist) – reverse the ordering of a list.

5.4.2 Rules for the library

Rules of guidelines have be defined in building up the library.

Rule 1 Set of elements is represented by list in Prolog. In CLP(R), a set or tuple is enclosed by " [" and "] ". For example, a set of binary relation (x,y) can be represented by:

[[a, 1], [b, 2], [c, 3], [d, 4]]

where $x = \{a, b, c, d\}, y = \{1, 2, 3, 4\}$

Rule 2 The order of list must be ignored for a set definition. But Prolog/CLP(R) will take care of the ordering, therefore, a predicate "equal(A, B)" will check the equality of the two lists.

```
equal([],[]).
equal([X|TX],B):-
   member(X,B), delete(X,B,BB),
    equal(TX,BB),!.
```

Rule 3 The list must contain no duplicates for a set definition except bags or sequences. A predicate rm_dup is constructed to remove duplicated elements in the list.

```
rm_tail(X,[],[]).
rm_tail(X,[X|Tail],List):-
       rm_tail(X,Tail,List).
```

```
rm_tail(X,[Y|Tail],[Y|List]):-
        not X == Y,
        rm_tail(X,Tail,List).

rm _dup([],[]).
rm_dup([X|TX],[X|TY]):-
      rm_tail(X,TX,TT),
      rm_dup(TT,TY).
```

Rule 4 The flow mode of arguments must be maximized if possible. In designing a predicate, an argument can be treated as input(i) or output(o) because we do not know the final usage of it. There are two strategies in facing with this problem, one is passive and one is active.

For the passive mode, it is the normal Prolog writing of [*Head* | *Tail*] for list. Prolog will backtrack the list instantiation for the required component. A typical example is:

```
% ii, oi
member(X,[X|_]).
member(X,[_|Y]):-  member(X,Y).
```

A query of member(X, [a, b, c, d]) will generate X equals to a, b, c, d respectively.

This method, however, is not very practical and efficient as huge logical combinations may come out. It is the case for set operations, set relations and mapping functions. In addition, the uses of *findall* and *setof* in Prolog will prohibit the reverse mode. For example, the set operation *union* can be defined as:

```
union(A,B,C):- setof(X,(member(X,A); member(X,B),C)).
```

Although the predicate is precise and concise, the flow mode of the arguments is only (iio).

For the active mode, the built-in function of CLP(R), var(?X) and nonvar(?X) are used to test the argument whether it is an unknown variable or has been instantiated a value. Different queries will be assigned to different predicates. Revisit the *union* example:

```
%flow mode can be iii, iio, oii, ioi
union(A,B,C):-
     nonvar(A), nonvar(B),
     setof(X, (member(X,A); member(X,B)), CC),!,
     equal(CC,C).
union(A,B,C):-
     nonvar(C), nonvar(B),
     difference(C,B,AA),
     subset(BB,B),
     union(AA,BB,A).
```

```
union(A,B,C):-
        nonvar(C),nonvar(A),
        difference(C,A,BB),
        subset(AA,A),
        union(BB,AA,B).
```

The testing results:

```
1 ?- [zclib].
*** Yes

2 ?- union([a,b],[b,c,d],R).
R = [a, b, c, d]
*** Yes

3 ?- union(R,[b,c],[a,b,c,d]).
R = [a, d]
*** Retry? ;
R = [a, d, b]
*** Retry? ;
R = [a, d, b, c]
*** Retry? ;
R = [a, d, c]
*** Retry? ;
*** No
```

```
4 ?- union([a,b],R,[a,b,c,d]).
R = [c, d]
*** Retry? ;
R = [c, d, a]
*** Retry? ;
R = [c, d, a, b]
*** Retry? ;
R = [c, d, b]
*** Retry? ;
*** No
```

Rule 5 Cut (!) is used very carefully to prevent unnecessary backtrack. Normally, the place of cut is tested by enforcing the "Retry?" question when the predicate is input with different data sets.

Rule 6 Predicates of defining number types will not generate possible numbers for solution. It is because mathematical equations can be handled by CLP(R). Integers, natural numbers, and real numbers are defined by:

```
%i
integer(X):-
   nonvar(X)->floor(X,X).
```

```
%i
natural(X):-
nonvar(X)->integer(X);
                X>=0.
```

```
%i
realno(X):-
   nonvar(X) -> real(X).
```

Here, the symbol $X->Y;Z$ is read as "if X then Y else Z". The function *floor*(X,X), *real*(X) are built-in functions of CLP(R) and will only be executed when X value is grounded. For natural numbers, if the value of X is unknown, $X>=0$ is used. The reason is that CLP(R) may not solve X yet and we may get the value by backtracking. Without nonvar(X), it will abort the execution by *floor*(X,X) and hence decrease the flexibility.

With the same reason, variables of type real numbers R are only checked when it is grounded.

5.4.3 Limitation of the Library

We can not expect each of the predicates has all the flow modes combination. The result may be an infinite set such as *intersect*(*What*, $[a,b]$, $[a]$). The set of "What" can be any set provided "a" is an element and b is not.

From the book of Z notation [35], some functions are not implemented. They are:

1. $\_^+$ Transitive closure $R^+ = \bigcup\{k : \mathbb{N}_1 \bullet R^k\}$ where R is a relation
2. $\_^*$ Reflexive-transitive closure $R^* = \bigcup\{k : \mathbb{N} \bullet R^k\}$
3. $\mathbb{F}$ Finite sets
4. $\mathbb{F}_1$ Non-empty finite sets
5. $\nrightarrow$ Finite partial functions

6. $\rightarrowtail\!\!\!\!\!\!+$ Finite partial injections

Items 1 and 2 may involve infinite sets and are not implemented. The remaining will duplicate some predicates such as partial functions, where we have assumed the inputs and outputs are finite list for any set functions or relations.

5.5 Z to CLP(R) Translator (ZCGEN)

The translator is a line interpretation process. The first scanning of the file will identify some basic entities such as dom, $\longrightarrow$, {, ... etc.. According to the type of entities, appropriate translation format is found from the rulebase. Figure 5.4 summarizes the flow of translation.

There are six kinds of entities:

1. Special words in Z. e.g. dom, ran, succ...
2. Special symbols in Z. e.g. $\longrightarrow, \rightarrowtail\!\!\!\!\!\!+$
3. Mathematical operators. e.g. sin, cos, ...
4. Normal words. e.g. title, variables.
5. Schema drawing lines. e.g. $\lceil, |$
6. Punctuation. e.g. $(), \{\}[\,]\,' \,?\,!$

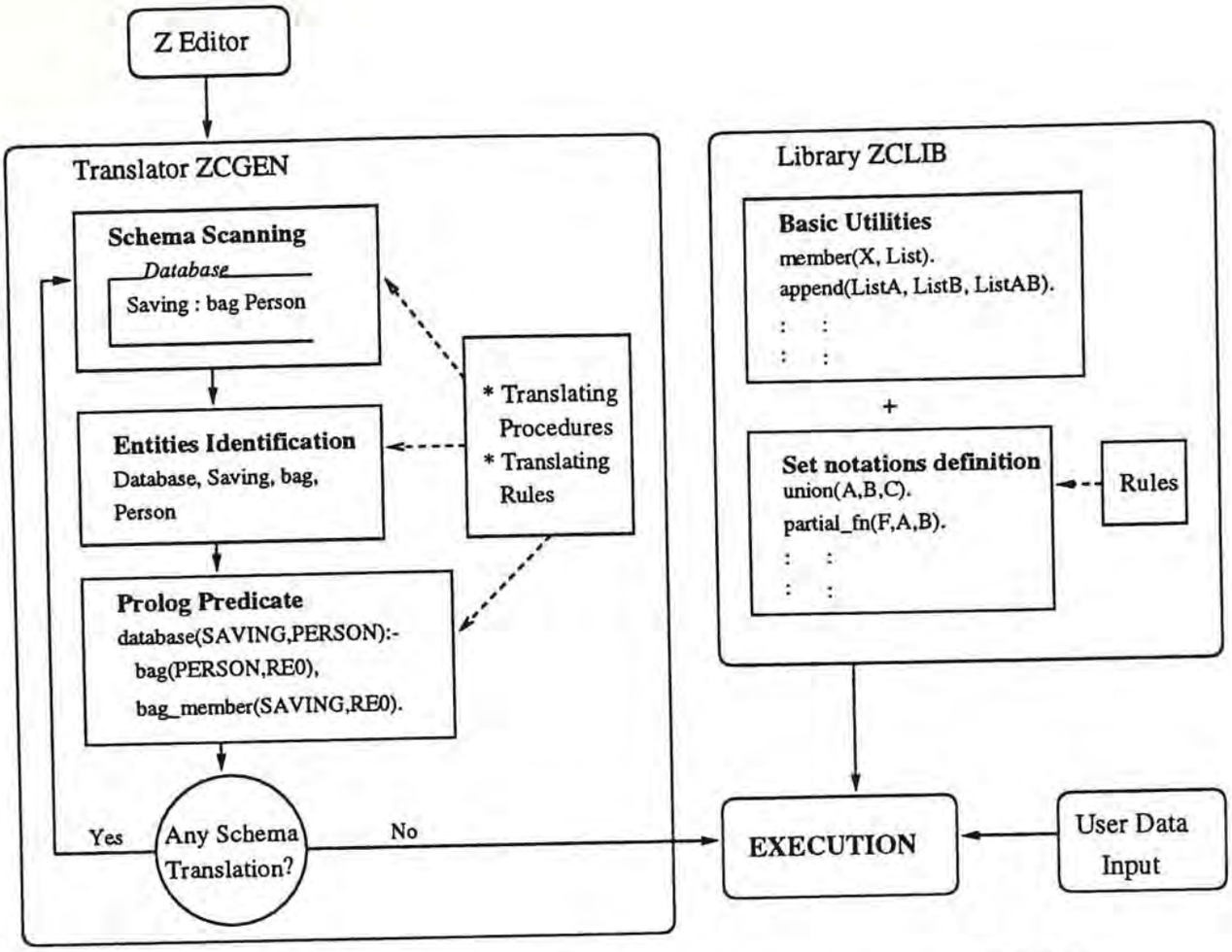


Figure 5.4: The Flow of the Prolog Translating system ZCGEN

5.5.1 Procedure for translation

Step 1 Scan the schema title.

Step 2 Scan the declaration one line at a time. Record the variables, types and determine any schemas reference. Numbers and unknowns will be identified.

Step 3 If there is a schema reference, the required predicate is searched out

from the database. All related arguments are passed into this predicate. Also, any state change, schema conjunction $\wedge$ and schema disjunction $\vee$ are determined.

Step 4 For the type definition, transform any relation in bracket and then transform any special keywords into Prolog predicates such as dom, ran, etc.

Step 5 Relate the variables with the type transformed or not.

For example, happy : $(X \times Y) \longrightarrow dom(\text{Relation})$

will be translated into

cart_prod(X,Y,RE0), dom(RELATION,RE1),
total_fn(HAPPY,RE0,RE1),

Step 6 If there are variables of type object class, the required class predicate is searched out from the schema database. All arguments in the object-class are headed by the present variables. Revisit the example in section 5.1:

```
e.g. schemaA(A1, A2, TypeA1, TypeA2):-
                A1 >= 10,
                A2 >= 20.

     schemaB(B1_A1, B1_A2):-
                B1_A1 >= 30,
```

```
B1_A2 = B2_A1 - 40,
schemaA(B1_A1, B1_A2, TypeA1, TypeA2).
% the last predicate checks the class properties
```

Step 7 Repeat from step 2 until declaration end.

Step 8 Scan the schema predicate part and do similar transformations as in steps 3, 4 and 5.

Step 9 Record this translated predicate's title and arguments.

5.5.2 Demonstration

A file update schema described in [24] and *Specification Case studies*[12] p.7 is :

FileUpdate

$$f, f' : Key \pfun Record$$
$$d? : \mathbb{P}\ Key$$
$$u? : Key \pfun Record$$

$$d? \subseteq \mathrm{dom}(f) \land$$
$$d? \cap \mathrm{dom}(u?) = \{\} \land$$
$$f' = (d? \ndres f) \oplus u?$$

After processed by ZCGEN, the output is:

```
fileupdate(F, F_pi, D_in, U_in, KEY, RECORD):-
  partial_fn(F, KEY, RECORD),
  subset(D_in, KEY),
  partial_fn(U_in, KEY, RECORD),
  dom(F, RE0),  subset_equ(D_in, RE0),
  dom(U_in, RE1),  intersect(D_in, RE1, []),
  dom_anti_restr(D_in, F, RE2),  override(RE2, U_in, F_pi),
  partial_fn(F_pi, KEY, RECORD),
  true.
```

We can find that the system ZCGEN translates the schema's semantic in a one-to-one mapping. Actually, the process is based on the following rules.

5.5.3 Rules for translation

Rule 1 Those variables involve input, output state changes and object class will be rewritten as:

Original Text	*Translated Text*
Var'	Var_pi
Var?	Var_in
Var!	Var_out
Var.class	Var_class

Rule 2 Words of mathematical operators such as sin, cos, ... are remained in

lower case spelling. Most Prolog systems have these mathematical functions.

Rule 3 In schema's declaration, those translations involve Var_pi and Var_out will be placed at the end of the predicate. This is because methods of final state or output value will only be defined in schema's predicates. Placing them at the end will act as a further confirmation. The original positions, however, may force unnecessary instantiation and result in backtracking. It can be found that the statement "partial_fn(F_pi, KEY, RECORD)" appeared in the previous demonstration is positioned at last. This method has been applied in [24] which is called "predicate promotion".

Rule 4 The class checking of new variables is added at the end of predicate. It is because the new constraints may determine the values of variables. The final placement has the same effect of rule 3.

Rule 5 Temporary transformation result will be stored as RE0, RE1, RE3 ... automatically. They act as the bridge elements.

Rule 6 If a predicate's line do not contain any set notations and schema calculus, it is normally a mathematical equation. The whole line will be copied and handled by CLP(R).

Rule 7 Powerset function e.g. $C : \mathbb{P} A$, will be translated into subset(C, A). Formally, the translation will be powerset(A, RE0), member(A, RE0) where RE0 is the set of all subsets of A. As the concern is only the membership property, it is not necessary to generate all the subsets of A. This method is same as [24].

Rule 8 For schema calculus, the operands of the referred schema will be passed into the present schema. Schema decoration consists of $schemaA'$, $\Delta schemaA$ and $\Xi schemaA$. Symbol $\Xi schemaA$ will just call out the $schemaA$. $\Delta schemaA$ is equivalent to $schemaA$ and $schemaA'$. Those variables in $schemaA'$, however, will be attached an extension of _pi to specify out a changing state. With the same reason of Rule 3, the calling of $schemaA'$ will be placed at the end of predicate. Schema conjunction $\wedge$ will be translated into " ," while disjunction $\vee$ is " ;".

Rule 9 For generic schema, the type of the generic parameters would be determined outside the schema.

5.5.4 Limitations of the Translator

During translation, multiple brackets interpretation have not been implemented. For example, $(X \rightarrow Y) \rightarrow Z$ is fine but not for $((X \rightarrow Y) \rightarrow Z)$. On the other hand, only one " = " sign is permitted in a line due to the simplicity of program design.

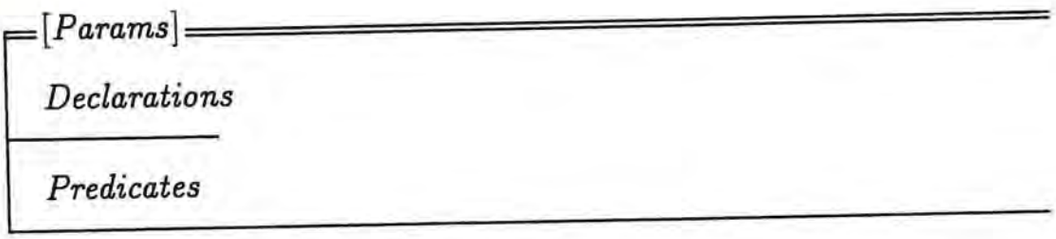
Quantifier statements, $\forall x... \bullet ...$ and $\exists x... \bullet ...$ are not implemented yet. It can be defined similar to the method found in *SuZan* [22] where a predicate "*for_all*(*Generator*, *Predicate*)" is built. But this method may generate numerous data which is extremely inefficient. In West and Eaglestone[39], predicate "*check_all_p*($[X \mid Y]$) : $-p(X)$, *check_all_p*(Y). *check_all_p*($[]$)." is proposed but it only works for finite known list. If we have the case $\forall x : Real \bullet z = x * y$, perhaps it is better to translate it into $z = x * y$ only. On the other hand, literature [10] has shown that existential quantifier such as $\exists x : T \bullet x > y$ can be omitted in logic specification languages because the statement $x > y$ has implicitly quantified the existence of x.

Some of the schema calculus described in [29] are not studied yet. They are:

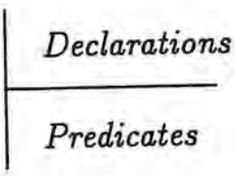
- Schema negation involves schema normalization and changing connectives.
- Schema hiding operators involve using universal quantifier.
- Schema composition involves passing the final state of a schema to the beginning state of another.

Schema without a title such as generic constant and axiomatic description is not allowed.

Generic definition



Axiomatic description



For translation purpose, they may need a particular title generated by the system.

5.6 Z to LaTeX translator (ZLATEX)

The purpose of this translator is to output Z schema in LaTeX format for printing and type checking with the use of *Fuzz* system. For printing, the document should include the [fuzz] documentstyle in the heading of file and then perform normal LaTeX compiling. For type checking, the command "fuzz example.tex" is used. Details of the *Fuzz* system usage can be found in the reference manual [36].

The translation procedure is similar to ZCGEN but now it is a one pass translator. Different entities will be identified and rewritten immediately according to the rulebase. The rulebase will produce the corresponding Fuzz's command codes. Unlike ZCGEN, it is not necessary to concern the relationships between

entities. In other words, its nature is like a look up table. For example, the schema described in section 5.5.2 will be translated to

```
\begin{schema}{FileUpdate}
f, f' : Key\pfun Record \\
d? : \power Key \\
u? : Key\pfun  Record
\where
d? \subseteq \dom(f)\land  \\
d?\cap \dom(u?) = \{\} \\
f'=(d?\ndres f)\oplus u?
\end{schema}
```

Chapter 6

Examples

Two examples will be shown, one is bags oriented and one is calculation oriented. They are self contained. A common characteristic is that real numbers calculation is involved.

6.1 A Simple Banking System

6.1.1 Bags

The system is similar to the one described in literature [11] where bag operations are used.

From Z notation [35], the definition of a bag is:

$$\text{bag } X == X \pfun \mathbb{N}_1.$$

We can find that a bag is a set of ordered pair:

$$\{X_1 \mapsto k_1, ..., X_n \mapsto k_n\} \text{ or written as } [\![X_1, ..., X_n]\!].$$

The domain of bags is the element while the range is their corresponding occurrences. Zero occurrence is not defined in a bag. We can say a set is a subset of a bag because each element only appears once. Similar to set, some bag operations such as bag membership $\sqsubseteq$, bag union $\uplus$ are defined in [35]. They have been implemented in the library ZCLIB.

Suppose we have a bag named "Alphabet" which contain elements [a, b, c, d, a, d]. Then, the library ZCLIB will output the predicate bag(Alphabet, Result) as:

$$\text{Result} = [[\text{a}, 2], [\text{b}, 1], [\text{c}, 1], [\text{d}, 2]].$$

One advantage of using bags over sets is the simplification. As mentioned in [11], we have a bag operation about the final balance after the transaction from one person to another:

$$Balance' = Balance \uplus \{from? \mapsto -Amount?, to? \mapsto Amount?\}$$

can be represented by set

$$Balance' = Balance \oplus \{from? \mapsto Balance(from?) - Amount?,$$
$$to? \mapsto Balance(to?) + amount?\}$$

where the symbol $\oplus$ is defined as

$$Q \oplus R = ((domR) \lhd Q) \cup R.$$

Here, symbol $\lhd$ denotes domain anti-restriction. The above semantic is the set of R union everything of Q except those elements appeared in R's domain.

The Balance relation will not be well defined in set if $from? = to?$, that is the case of same person. It is because we do not know the value of balance(to?) is taken from before or after the $bank(from?) - Amount?$ operation. However, bags can allow the summation of frequencies over the same object.

Literature [11] has also generalized the bags that frequency of occurrence can be negative, as well as the usual zero. We further extend the idea to include the infinite set of real numbers. Hence,

$$\text{bag X} == \text{X} \pfun \text{Real number}$$

6.1.2 Specifications

Suppose the Bank has two simple services, manual transactions and auto-transfer-machine. Then, we define two sets

$[Person, Real]$

Person is the set of customers and *Real* is the set of real numbers. The saving of *Person* will be represented by the frequency of occurrences, i.e. bag *Person*.

Database

$Saving : \text{bag } Person$

transfer[*Person*]

$Tbalance! : \text{bag } Person$

$From?, To? : Person$

$Amount? : Real$

$Tbalance! = \{(From? \mapsto -Amount?), (to? \mapsto Amount?)\}$

atm[*Person*]

$Abalance! : \text{bag } Person$

$Name? : Person$

$Withdraw? : Real$

$Abalance! = \{(Name? \mapsto -Withdraw?)\}$

balance

$\Delta Database$

$transfer[Person]$

$atm[Person]$

$changes! : \text{bag } Person$

$changes! = Tbalance! \uplus Abalance!$

$Saving' = Saving \uplus changes!$

Let the set *Person* has three members peter, mary, john with saving 2000, 3000, 400 respectively. We would like to know the final balance if peter transfers 1234.5 dollars to john and mary withdraws 680.5 dollars from atm.

After the automatic translation by ZCGEN, the Prolog predicates are:

```
database(SAVING, PERSON):-
  bag(PERSON, RE0),  bag_member(SAVING, RE0),
  true.

transfer(TBALANCE_out, FROM_in, TO_in, AMOUNT_in, PERSON):-
  bag(PERSON, RE1),  member(FROM_in, PERSON),  member(TO_in, PERSON),
  realno(AMOUNT_in),
  maplet(FROM_in, -AMOUNT_in, RE2),  maplet(TO_in, AMOUNT_in, RE3),
  equal(TBALANCE_out, [RE2,RE3]),
  bag_member(TBALANCE_out, RE1),
  true.

atm(ABALANCE_out, NAME_in, WITHDRAW_in, PERSON):-
  bag(PERSON, RE4),  member(NAME_in, PERSON),
  realno(WITHDRAW_in),
  maplet(NAME_in, -WITHDRAW_in, RE5),  equal(ABALANCE_out, [RE5]),
  bag_member(ABALANCE_out, RE4),
  true.

balance(CHANGES_out, SAVING, PERSON, SAVING_pi, TBALANCE_out, FROM_in,
  TO_in, AMOUNT_in, ABALANCE_out, NAME_in, WITHDRAW_in):-
  database(SAVING, PERSON),
  transfer(TBALANCE_out, FROM_in, TO_in, AMOUNT_in, PERSON),
  atm(ABALANCE_out, NAME_in, WITHDRAW_in, PERSON),
```

```
  bag(PERSON, RE6),  bag_union(TBALANCE_out, ABALANCE_out, CHANGES_out),
  bag_union(SAVING, CHANGES_out, SAVING_pi),
  database(SAVING_pi, PERSON),
  bag_member(CHANGES_out, RE6),
  true.

go :-
  SAVING = [[peter,2000], [mary,3000], [john,400]],
  PERSON = [peter, mary, john],
  FROM_in = peter,
  TO_in = john,
  AMOUNT_in = 1234.56,
  NAME_in = mary,
  WITHDRAW_in = 680.5,
  balance(CHANGES_out, SAVING, PERSON, SAVING_pi, TBALANCE_out, FROM_in,
  TO_in, AMOUNT_in, ABALANCE_out, NAME_in, WITHDRAW_in),
  write('The answer of CHANGES_out is '),
  writeln(CHANGES_out),
  write('The answer of SAVING_pi is '),
  writeln(SAVING_pi),
  write('The answer of TBALANCE_out is '),
  writeln(TBALANCE_out),
  write('The answer of ABALANCE_out is '),
  writeln(ABALANCE_out),
  true.
```

The data inside the go:- predicate is input by user. Afterwards, the running process and result will be:

```
CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991, 1992) All Rights Reserved

1 ?- [zclib].

*** Yes

2 ?- [bank_p].

*** Yes

3 ?- go.
The answer of CHANGES_out is [[peter, -1234.56], [john, 1234.56],
[mary, -680.5]]
The answer of SAVING_pi is [[peter, 765.44], [mary, 2319.5],
[john, 1634.56]]
The answer of TBALANCE_out is [[peter, -1234.56], [john, 1234.56]]
The answer of ABALANCE_out is [[mary, -680.5]]

*** Retry? y
```

```
*** No

4 ?-
```

From the above, line 1 is loading the library ZCLIB and line 2 is loading the translated predicates of the bank system. Line 3 is the execution results.

6.2 A Graphics Example

We are going to specify a drawing of trolley on the X-Y plane. The trolley consists of a body and two wheels as shown in figure 6.1. The approach is to use object class definition.

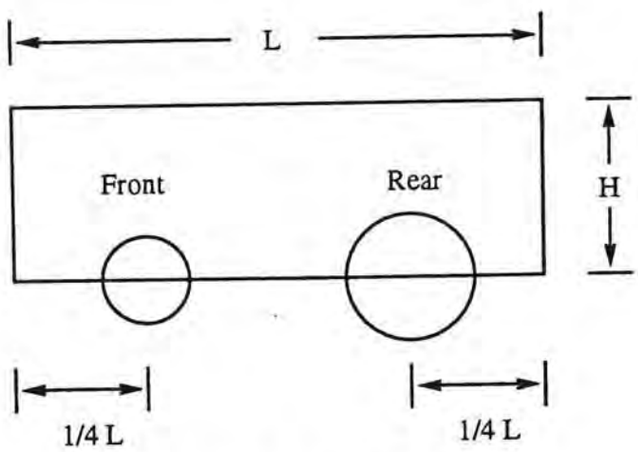


Figure 6.1: A Trolley

6.2.1 Defining a Rectangle

The body is a rectangle which belongs to the class of quadrilateral. The class of quadrilateral consists of four vectors. In figure 6.2, each vector has its own magnitude and angle. Since the vectors must end in a loop, the summation of their x-y components should be zero.

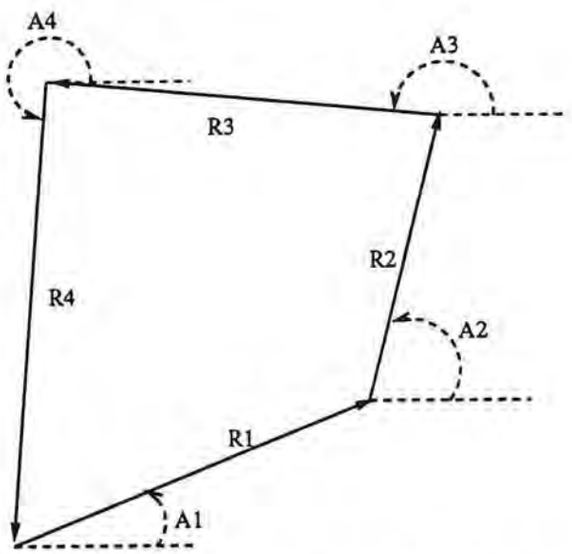


Figure 6.2: Defining a quadrilateral in the vector space

Hence we have the equation:

$$R1*\cos(A1) + R2*\cos(A2) + R3*\cos(A3) + R4*\cos(A4) = 0$$
$$R1*\sin(A1) + R2*\sin(A2) + R3*\sin(A3) + R4*\sin(A4) = 0$$

Then, the rectangle is a quadrilateral whose opposite sides are equal in length and one angle is a right angle.

$$R1 = R3, \quad R2 = R4, \quad A2\text{-}A1 = \pi / 2$$

6.2.2 Drawing a Rectangle

When we want to draw a rectangle on a x-y plane, it may involve the rotation and movement. It is easy to rotate the rectangle in vector space by adding each angle with the amount to be rotated. On the other hand, the moving process will need an initial coordinate. Suppose the corner (X1, Y1) is situated at the origin (0,0) and lying horizontal. The remaining corners are constrained by the lengths and angles. Then, the moving results will be calculated by adding the new positions (X1', Y1') to each corner. Figure 6.3 shows the rotation and movement process.

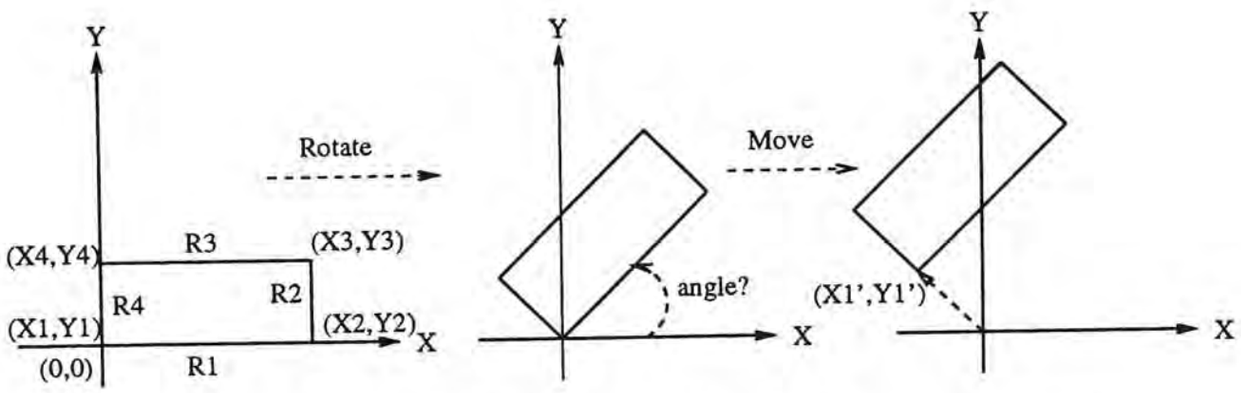


Figure 6.3: Drawing a rectangle with rotation and movement

6.2.3 Defining a Circle

A circle is characterized by center of coordinates, radius and circumference. We have the relation:

$$\text{Circumference} >= 0, \quad \text{Circumference} = 2 * \pi * \text{Radius}.$$

About the center, rotation is meaningless while translation is just done by changing its coordinate.

6.2.4 Specifications

Let the trolley's body is of size 15 x 7 units, lying horizontally with corner 1 at the origin. Circumference of front wheel is 15 and rear wheel is 24 units. We are interested in the final coordinates if we rotate the trolley $\pi/2$ counter-clockwise and moving corner 1 to point (7, 15) as shown in figure 6.4.

The drawing of wheels are related to the final position of the body. It is not necessary to perform moving from the origin as done in the case of body.

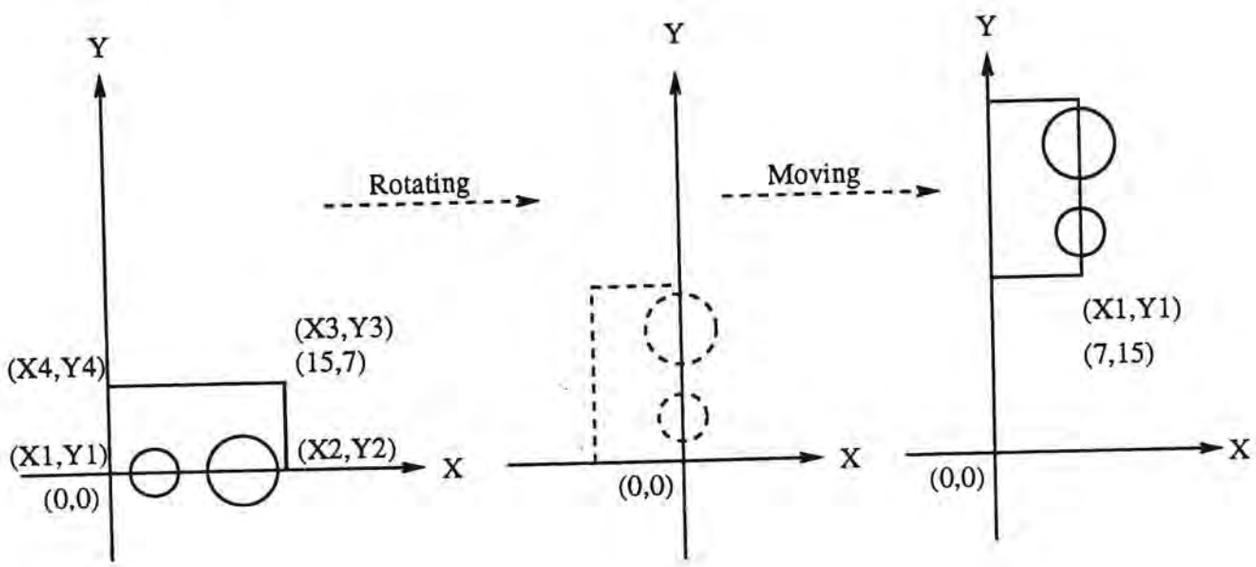


Figure 6.4: Drawing a Trolley

We define the type

$[Real]$

representing real number.

SystemError

$$Errlimit : Real$$

Quad

$$R1, R2, R3, R4, A1, A2, A3, A4 : Real$$

$$R1 * cos(A1) + R2 * cos(A2) + R3 * cos(A3) + R4 * cos(A4) = 0$$

$$R1 * sin(A1) + R2 * sin(A2) + R3 * sin(A3) + R4 * sin(A4) = 0$$

Circle

$$CenterX, CenterY, Radius, Circum : Real$$

$$Circum >= 0$$

$$Circum = 2 * 3.1416 * Radius$$

Rectangle

$$SystemError$$

$$Rect : Quad$$

$$Rect.R1 - Rect.R3 <= Errlimit$$

$$Rect.R2 - Rect.R4 <= Errlimit$$

$$Rect.A2 - Rect.A1 - 1.5708 <= Errlimit$$

Rotrec

$\Delta Rectangle$

$rotangle? : Real$

$Errlimit' = Errlimit$

$Rect.A1' = Rect.A1 + rotangle?, Rect.R1' = Rect.R1$

$Rect.A2' = Rect.A2 + rotangle?, Rect.R2' = Rect.R2$

$Rect.A3' = Rect.A3 + rotangle?, Rect.R3' = Rect.R3$

$Rect.A4' = Rect.A4 + rotangle?, Rect.R4' = Rect.R4$

Drawrect

$Rotrect$

$X1, X2, X3, X4, Y1, Y2, Y3, Y4 : Real$

$TranX?, TranY? : Real$

$X1 = TranX?, Y1 = TranY?$

$X2 = X1 + Rect.R1' * cos(Rect.A1'), Y2 = Y1 + Rect.R1' * sin(Rect.A1')$

$X3 = X2 + Rect.R2' * cos(Rect.A2'), Y3 = Y2 + Rect.R2' * sin(Rect.A2')$

$X4 = X3 + Rect.R3' * cos(Rect.A3'), Y4 = Y3 + Rect.R3' * sin(Rect.A3')$

Trolley

$Body : Drawrect$

$Fwheel, Rwheel : Circle$

$Fwheel.Circum = 15$

$Rwheel.Circum = 24$

$Fwheel.CenterX = Body.X1 + (Body.X2 - Body.X1)/4$

$Fwheel.CenterY = Body.Y1 + (Body.Y2 - Body.Y1)/4$

$Rwheel.CenterX = Body.X1 + (Body.X2 - Body.X1) * 3/4$

$Rwheel.CenterY = Body.Y1 + (Body.Y2 - Body.Y1) * 3/4$

After the automatic Prolog translation by ZCGEN, the predicates are :

```
systemerror(ERRLIMIT):-
  realno(ERRLIMIT),
  true.

quad(R1, R2, R3, R4, A1, A2, A3, A4):-
  realno(R1), realno(R2), realno(R3), realno(R4), realno(A1),
  realno(A2), realno(A3), realno(A4),
  R1*cos(A1)+R2*cos(A2)+R3*cos(A3)+R4*cos(A4)=0,
  R1*sin(A1)+R2*sin(A2)+R3*sin(A3)+R4*sin(A4)=0,
  true.
```

```
circle(CENTERX, CENTERY, RADIUS, CIRCUM):-
  realno(CENTERX), realno(CENTERY), realno(RADIUS),
  realno(CIRCUM),
  CIRCUM>=0,
  CIRCUM=2*3.1416*RADIUS,
  true.

rectangle(ERRLIMIT, RECT_R1, RECT_R2, RECT_R3, RECT_R4, RECT_A1,
  RECT_A2, RECT_A3, RECT_A4):-
  systemerror(ERRLIMIT),
  RECT_R1 -RECT_R3<=ERRLIMIT,
  RECT_R2 -RECT_R4<=ERRLIMIT,
  RECT_A2 -RECT_A1 -1.5708<=ERRLIMIT,
  quad(RECT_R1, RECT_R2, RECT_R3, RECT_R4, RECT_A1, RECT_A2, RECT_A3,
    RECT_A4),
  true.

rotrect(ROTANGLE_in, ERRLIMIT, RECT_R1, RECT_R2, RECT_R3, RECT_R4,
  RECT_A1, RECT_A2, RECT_A3, RECT_A4, ERRLIMIT_pi, RECT_R1_pi,
  RECT_R2_pi, RECT_R3_pi, RECT_R4_pi, RECT_A1_pi, RECT_A2_pi,
  RECT_A3_pi, RECT_A4_pi):-
  rectangle(ERRLIMIT, RECT_R1, RECT_R2, RECT_R3, RECT_R4, RECT_A1,
    RECT_A2, RECT_A3, RECT_A4),
  realno(ROTANGLE_in),
  ERRLIMIT_pi=ERRLIMIT,
```

```
 RECT_A1_pi=RECT_A1+ROTANGLE_in,RECT_R1_pi=RECT_R1,
 RECT_A2_pi=RECT_A2+ROTANGLE_in,RECT_R2_pi=RECT_R2,
 RECT_A3_pi=RECT_A3+ROTANGLE_in,RECT_R3_pi=RECT_R3,
 RECT_A4_pi=RECT_A4+ROTANGLE_in,RECT_R4_pi=RECT_R4,
 rectangle(ERRLIMIT_pi, RECT_R1_pi, RECT_R2_pi, RECT_R3_pi,
   RECT_R4_pi, RECT_A1_pi, RECT_A2_pi, RECT_A3_pi, RECT_A4_pi),
 true.

drawrect(X1, X2, X3, X4, Y1, Y2, Y3, Y4, TRANX_in, TRANY_in,
 ROTANGLE_in, ERRLIMIT, RECT_R1, RECT_R2, RECT_R3, RECT_R4, RECT_A1,
 RECT_A2, RECT_A3, RECT_A4, ERRLIMIT_pi, RECT_R1_pi, RECT_R2_pi,
 RECT_R3_pi, RECT_R4_pi, RECT_A1_pi, RECT_A2_pi, RECT_A3_pi,
 RECT_A4_pi):-
 rotrect(ROTANGLE_in, ERRLIMIT, RECT_R1, RECT_R2, RECT_R3, RECT_R4,
   RECT_A1, RECT_A2, RECT_A3, RECT_A4, ERRLIMIT_pi, RECT_R1_pi,
   RECT_R2_pi, RECT_R3_pi, RECT_R4_pi, RECT_A1_pi, RECT_A2_pi,
   RECT_A3_pi, RECT_A4_pi),
 realno(X1), realno(X2), realno(X3), realno(X4), realno(Y1),
 realno(Y2), realno(Y3), realno(Y4), realno(TRANX_in),
 realno(TRANY_in),
 X1=TRANX_in,Y1=TRANY_in,
 X2=X1+RECT_R1_pi*cos(RECT_A1_pi),Y2=Y1+RECT_R1_pi*sin(RECT_A1_pi),
 X3=X2+RECT_R2_pi*cos(RECT_A2_pi),Y3=Y2+RECT_R2_pi*sin(RECT_A2_pi),
 X4=X3+RECT_R3_pi*cos(RECT_A3_pi),Y4=Y3+RECT_R3_pi*sin(RECT_A3_pi),
 true.
```

```
trolley(BODY_X1, BODY_X2, BODY_X3, BODY_X4, BODY_Y1, BODY_Y2, BODY_Y3,
  BODY_Y4, BODY_TRANX_in, BODY_TRANY_in, BODY_ROTANGLE_in,
  BODY_ERRLIMIT, BODY_RECT_R1, BODY_RECT_R2, BODY_RECT_R3, BODY_RECT_R4,
  BODY_RECT_A1, BODY_RECT_A2, BODY_RECT_A3, BODY_RECT_A4,
  BODY_ERRLIMIT_pi, BODY_RECT_R1_pi, BODY_RECT_R2_pi, BODY_RECT_R3_pi,
  BODY_RECT_R4_pi, BODY_RECT_A1_pi, BODY_RECT_A2_pi, BODY_RECT_A3_pi,
  BODY_RECT_A4_pi, FWHEEL_CENTERX, FWHEEL_CENTERY, FWHEEL_RADIUS,
  FWHEEL_CIRCUM, RWHEEL_CENTERX, RWHEEL_CENTERY, RWHEEL_RADIUS,
  RWHEEL_CIRCUM):-
  FWHEEL_CIRCUM=15,
  RWHEEL_CIRCUM=24,
  FWHEEL_CENTERX=BODY_X1+(BODY_X2 -BODY_X1)/4,
  FWHEEL_CENTERY=BODY_Y1+(BODY_Y2 -BODY_Y1)/4,
  RWHEEL_CENTERX=BODY_X1+(BODY_X2 -BODY_X1)*3/4,
  RWHEEL_CENTERY=BODY_Y1+(BODY_Y2 -BODY_Y1)*3/4,
  drawrect(BODY_X1, BODY_X2, BODY_X3, BODY_X4, BODY_Y1, BODY_Y2,
    BODY_Y3, BODY_Y4, BODY_TRANX_in, BODY_TRANY_in, BODY_ROTANGLE_in,
    BODY_ERRLIMIT, BODY_RECT_R1, BODY_RECT_R2, BODY_RECT_R3,
    BODY_RECT_R4, BODY_RECT_A1, BODY_RECT_A2, BODY_RECT_A3, BODY_RECT_A4,
    BODY_ERRLIMIT_pi, BODY_RECT_R1_pi, BODY_RECT_R2_pi, BODY_RECT_R3_pi,
    BODY_RECT_R4_pi, BODY_RECT_A1_pi, BODY_RECT_A2_pi, BODY_RECT_A3_pi,
    BODY_RECT_A4_pi),
  circle(FWHEEL_CENTERX, FWHEEL_CENTERY, FWHEEL_RADIUS, FWHEEL_CIRCUM),
  circle(RWHEEL_CENTERX, RWHEEL_CENTERY, RWHEEL_RADIUS, RWHEEL_CIRCUM),
```

```
  true.

go :-
  BODY_TRANX_in = 7,
  BODY_TRANY_in = 15,
  BODY_ROTANGLE_in = 1.5708,
  BODY_ERRLIMIT = 0.001,
  BODY_RECT_R1 = 15,
  BODY_RECT_R2 = 7,
  BODY_RECT_A1 = 0,
  BODY_RECT_A2 = 1.5708,
  BODY_RECT_A3 = 3.1416,
  BODY_RECT_A4 = -1.5708,
  trolley(BODY_X1, BODY_X2, BODY_X3, BODY_X4, BODY_Y1, BODY_Y2, BODY_Y3,
    BODY_Y4, BODY_TRANX_in, BODY_TRANY_in, BODY_ROTANGLE_in,
    BODY_ERRLIMIT, BODY_RECT_R1, BODY_RECT_R2, BODY_RECT_R3, BODY_RECT_R4,
    BODY_RECT_A1, BODY_RECT_A2, BODY_RECT_A3, BODY_RECT_A4,
    BODY_ERRLIMIT_pi, BODY_RECT_R1_pi, BODY_RECT_R2_pi, BODY_RECT_R3_pi,
    BODY_RECT_R4_pi, BODY_RECT_A1_pi, BODY_RECT_A2_pi, BODY_RECT_A3_pi,
    BODY_RECT_A4_pi, FWHEEL_CENTERX, FWHEEL_CENTERY, FWHEEL_RADIUS,
    FWHEEL_CIRCUM, RWHEEL_CENTERX, RWHEEL_CENTERY, RWHEEL_RADIUS,
    RWHEEL_CIRCUM),
  write('The answer of BODY_X1 is '),
  writeln(BODY_X1),
  write('The answer of BODY_X2 is '),
```

```
writeln(BODY_X2),
write('The answer of BODY_X3 is '),
writeln(BODY_X3),
write('The answer of BODY_X4 is '),
writeln(BODY_X4),
write('The answer of BODY_Y1 is '),
writeln(BODY_Y1),
write('The answer of BODY_Y2 is '),
writeln(BODY_Y2),
write('The answer of BODY_Y3 is '),
writeln(BODY_Y3),
write('The answer of BODY_Y4 is '),
writeln(BODY_Y4),
write('The answer of BODY_RECT_R3 is '),
writeln(BODY_RECT_R3),
write('The answer of BODY_RECT_R4 is '),
writeln(BODY_RECT_R4),
write('The answer of BODY_ERRLIMIT_pi is '),
writeln(BODY_ERRLIMIT_pi),
write('The answer of BODY_RECT_R1_pi is '),
writeln(BODY_RECT_R1_pi),
write('The answer of BODY_RECT_R2_pi is '),
writeln(BODY_RECT_R2_pi),
write('The answer of BODY_RECT_R3_pi is '),
writeln(BODY_RECT_R3_pi),
```

```
write('The answer of BODY_RECT_R4_pi is '),
writeln(BODY_RECT_R4_pi),
write('The answer of BODY_RECT_A1_pi is '),
writeln(BODY_RECT_A1_pi),
write('The answer of BODY_RECT_A2_pi is '),
writeln(BODY_RECT_A2_pi),
write('The answer of BODY_RECT_A3_pi is '),
writeln(BODY_RECT_A3_pi),
write('The answer of BODY_RECT_A4_pi is '),
writeln(BODY_RECT_A4_pi),
write('The answer of FWHEEL_CENTERX is '),
writeln(FWHEEL_CENTERX),
write('The answer of FWHEEL_CENTERY is '),
writeln(FWHEEL_CENTERY),
write('The answer of FWHEEL_RADIUS is '),
writeln(FWHEEL_RADIUS),
write('The answer of FWHEEL_CIRCUM is '),
writeln(FWHEEL_CIRCUM),
write('The answer of RWHEEL_CENTERX is '),
writeln(RWHEEL_CENTERX),
write('The answer of RWHEEL_CENTERY is '),
writeln(RWHEEL_CENTERY),
write('The answer of RWHEEL_RADIUS is '),
writeln(RWHEEL_RADIUS),
write('The answer of RWHEEL_CIRCUM is '),
```

```
writeln(RWHEEL_CIRCUM),
true.
```

Under the CLP(R) system, the running process is:

```
CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991, 1992) All Rights Reserved

1 ?- [zclib].

*** Yes

2 ?- [troll_p].

*** Yes

3 ?- go.
The answer of BODY_X1 is 7
The answer of BODY_X2 is 6.99994
The answer of BODY_X3 is -5.50979e-05
The answer of BODY_X4 is 0.000110196
The answer of BODY_Y1 is 15
The answer of BODY_Y2 is 30
```

```
The answer of BODY_Y3 is 29.9999
The answer of BODY_Y4 is 15
The answer of BODY_RECT_R3 is 14.9999
The answer of BODY_RECT_R4 is 6.99989
The answer of BODY_ERRLIMIT_pi is 0.001
The answer of BODY_RECT_R1_pi is 15
The answer of BODY_RECT_R2_pi is 7
The answer of BODY_RECT_R3_pi is 14.9999
The answer of BODY_RECT_R4_pi is 6.99989
The answer of BODY_RECT_A1_pi is 1.5708
The answer of BODY_RECT_A2_pi is 3.1416
The answer of BODY_RECT_A3_pi is 4.7124
The answer of BODY_RECT_A4_pi is 0
The answer of FWHEEL_CENTERX is 6.99999
The answer of FWHEEL_CENTERY is 18.75
The answer of FWHEEL_RADIUS is 2.38732
The answer of FWHEEL_CIRCUM is 15
The answer of RWHEEL_CENTERX is 6.99996
The answer of RWHEEL_CENTERY is 26.25
The answer of RWHEEL_RADIUS is 3.81971
The answer of RWHEEL_CIRCUM is 24

*** Yes
```

We find that the results agree with our expectation.

Body Expected:	*Calculated*
(X1, Y1) = (7, 15)	(7, 15)
(X2, Y2) = (7, 30)	(6.99994, 30)
(X3, Y3) = (0, 30)	(-5.50979e-05, 29.9999)
(X4, Y4) = (0, 15)	(0.000110196, 15)
Wheels Expected :	
(FW_X, FW_Y) = (7, 18.75)	(6.99999, 18.75)
(RW_X, RW_Y) = (7, 26.25)	(6.99996, 26.25)
FW radius = 2.387324146	2.38732
RW radius = 3.819718634	3.81971

6.3 Specifications Writing Experience

The examples raised in the last section involve the following characteristics:

- Bags operation
- Schema Inclusion
- Schema decoration or changing state
- Object class type
- Real numbers calculation

The schemas' writing approach is in an expanding order. The basic and simplest schema is placed at the beginning while the final executing control schema

is placed at last. The reason is for the Prolog execution which is a depth first search algorithm.

It is found that all the examples fail the type checking by the Fuzz system. The main reason is the lack of real numbers definition in conventional Z. Although we define the set [Real], *Fuzz* does not realize the nature of real numbers. It will treat it as a normal set. Therefore, the variable "-Amount?" would be classified as an undefined type. In addition, Fuzz does not have the knowledge of mathematical functions (sin, cos, ...).

Nevertheless, a correct execution result is a strong evidence of right specifications. Experience has shown that any typing mistakes or wrong equations in the schema would produce an unpredicatable response. This feedback is quick and solid.

Furthermore, it is found interesting to define a system error in the drawing example. Without this schema, CLP(R) would response "No" during execution even the specification is absolutely correct. It is because there is rounding error of the CLP(R). Results of calculation may not agree or equal to the expected value. The precision of the present system is about six decimal digits.

Inspecting the Prolog predicates, the arguments of predicates are increasing. They are the schema inclusion, decoration, object class bringing their arguments to the present predicate. On the other view, it is tedious to input or investigate

such long predicates. Thus, we can say that the Z schema can act as a documentation of the related Prolog predicates.

Chapter 7

Conclusion

7.1 Contributions

In this thesis, we have presented a system called ZCLP(R) that can execute Z by translating Z schemas into executable Prolog predicates. The system can handle the real numbers type which is undefined in conventional Z. To the best of our knowledge, this is the first attempt to extend the animation of Z into the area of continuous mathematics. It is feasible because we have used the CLP(R) system where real number calculations are handled by delayed evaluation and constraint satisfaction. Unlike standard Prolog, users do not need to take care of the numbers' generation for constraints satisfaction which may be an infinite process.

ZCLP(R) has the following characteristics:

1. The translation is simple because both languages are based on first order

logic. We can perform a one-to-one mapping of the semantics. We can say Logic Programming and Z are more or less at the same abstract level. Although there is no formal proof of the correctness of this translation, an animation result can really feedback a behavioural model.

2. The translation is rapid and solid. On average, the translation process only takes a few seconds on a 486 based computer. The translated file is then loaded into a DEC workstation. Together with CLP(R) and the library ZCLIB, the animating process takes less than a second. Such rapid feedback can also achieve correct specifications which are assumed by comparing the results with manual checking. Numerical agreement will further enrich the specifier's confidence.

3. The translation requires data refinement. From the examples, we have found the specifications are from simple to complex. It is because Prolog is of depth first search, it is convenient to write down the basis schemas first and then perform the schema calculus or object orientation. In addition, a control schema is needed to start the whole animation. Normally, it is placed last so that the called schemas have been translated and recorded by Prolog.

Overall, ZCLP(R) consists of four subsystems, namely:

- *ZEDIT* - Z editor.

- *ZCGEN* - Prolog translator.
- *ZCLIB* - Set notations' functions library.
- *ZLATEX* - LaTeX translator.

The ZEDIT is an user friendly editor which can generate set notations and schema lines on screen.

Example in the demonstration section of chapter 5 has shown that the ZCGEN system can do the same translation result as appeared in literature [24]. Besides, chapter 6 has shown the system ZCLP(R) is capable handling real number calculation, schema inclusion, schema decoration, concepts of bags and simple object oriented. These powers are very useful in the current development of Z. They are never been integrated into a single system among the pervious literatures. Therefore, we can say ZCLP(R) is a more completed translating system than before.

The library ZCLIB is built to translate the set notations' functions in Z. Efforts have been paid in trying to build all the notations appeared in [35]. A major problem is about the flow modes of the predicate's arguments. In general, they can be used as both input and output modes. It is considered in chapter 5 where methods of solution are suggested. The result also reduces the execution time. Table 7.1 shows out the summary of the translatable notations.

Another translator Z to LaTeX has also been developed. It can translate Z

Basic expression				
$\neg$	$\wedge$	$\vee$	$=$	$\neq$
Δ	Ξ	$\mathbb{N}$	$\mathbb{N}_1$	$\mathbb{Z}$
min	*max*	succ	*Real*	
Sets				
$\in$	$\notin$	$\subseteq$	$\subset$	$\mathbb{P}$
$\mathbb{P}_1$	$\cup$	$\cap$	$\bigcup$	$\bigcap$
$\varnothing$	$\times$	..		
Relations				
$\leftrightarrow$	$\mapsto$	dom	ran	id
$\lhd$	$\rhd$	$\lhd$	$\rhd$	$(\!\mid\!)$
$\fatsemi$	$\circ$	$\oplus$		
Functions				
$\pfun$	$\fun$	$\rightarrowtail$	$\mapsto$	$\twoheadrightarrow$
$\surj$	$\mapsto$			
Sequences				
seq	seq_1	iseq	$\frown$	$\upharpoonright$
$\uparrow$	$\frown/$	*rev*	*head*	*last*
tail	*front*	*squash*	prefix	suffix
in	disjoint	partition		
Bags				
bag	*count*	*items*	$\uplus$	$\biguplus$
$\otimes$	$\sqsubseteq$	$\sqsubseteq$	*bags multiplication*	
distributed bag addition				

Table 7.1: Summary of set notations recognized by system ZCLP(R)

schemas into LaTeX format with the command codes recognized by the *Fuzz* system, through which, we can do the printing and type checking. Unfortunately, the type checking is not available for real numbers because it is undefined in conventional Z.

7.2 Difficulties

The main difficulty of the research is the Z language itself. It is a new formal method language and it has not been standardized yet. New writing style (in case of object oriented) or notations are proposed from time to time. In addition, there is a lack of supporting tools. In this thesis, it is found the tools are built from the beginning of editor.

As Z uses set theory and notations to express the specifications, variations of this abstract writing causes difficulty for the translator to interpret. For example, we can write a predicate in one line such as

$$(A \in B) \land (B \subset C)$$

or in two lines

$$A \in B\,,$$
$$B \subset C\,.$$

Therefore, we have the limitations in writing as described in chapter 5. In other words, more intelligent translator is needed. Also, it is very difficult to

write a good specifications. The difficulty is increased when we consider the possibility of translating into Prolog. It is restricted by ZCGEN and ZCLIB.

Follower of this research must be familiar with Prolog. It is a language that careful logical thinking is needed when we use the techniques of backtracking. It is a powerful language, however, with a disadvantage that the documentation is very bad. Others will find it very difficult to follow or appreciate one's Prolog program.

The CLP(R), a special extension of Prolog, uses concepts of delayed evaluation and constraint satisfaction to solve the real numbers calculation. It is a newly developed software that the built in functions are limited. Unlike other Prolog, we have to create these functions by ourselves.

In building up the library, time is consuming in testing and rewriting for the general cases of the predicates in order to maximize the flow modes.

7.3 Further Works

Up to now, the system involves operations on workstation and PC. For consistency, a pure PC system or a pure workstation system is needed. The present method is based on workstation. A window is opened to simulate a PC DOS environment on the DEC workstation, and a window is opened to operate the CLP(R) system.

There are still some further works needed for the system ZCLP(R). First, a more user friendly editor is needed. Its operations can be mouse driven. Second, section 5.5.4 has described some limitations of the translator ZCGEN. For example, we have to interpret some complex predicates or to include the universal quantifier statement. These can be improved. Third, an intelligent user interface is needed for data input. It can check any syntax or type errors.

In section 3.3, we have mentioned that real number calculations should be free of precision problem. One solution is to use lists to represent numbers. We can then perform arithmetic by list operations. The system CLP(R), however, does not apply its delayed evaluation to strings of list. The accuracy of present model is machine dependent. Therefore, it is valuable to develop a system similar to CLP(R) but with real numbers implemented as lists.

Bibliography

[1] Ivan Bratko. *Prolog, programming for artificial intelligence.* Addison-Wesley, second edition, 1990.

[2] David Brownbridge. Using z to develop a case toolset. In J. E. Nicholls, editor, *Z User workshop, Oxford 1989*, pages 142–149. Springer-Verlag, 1989.

[3] Claude W. Burrill. *Foundations of Real Numbers.* McGraw-Hill, 1967.

[4] B. Cohen. Justification of formal methods for system specification. *Software Engineering Journal*, pages 26–35, January 1989.

[5] B. Cohen. A rejustification of formal notatioins. *Software Engineering Journal*, pages 36–38, January 1989.

[6] Roger Duke Paul King Gordon Rose Graeme Smith David Carrington, David Duke. Object-z: An object-oriented extension to z. In S. T. Vuong, editor, *Formal Description Techniques, II*, pages 281–296. Elsevier Science Publishers B. V. (North-Holland), 1990.

[7] Vernoika Doma and Robin Nicholl. Ez: A system for automatic prototyping of z specifications. In W. J. Toetenel S. Prehn, editor, *VDM '91: Formal*

Software development methods: 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991 Proceedings, pages 189–203, Berlin, 1991. Springer-Verlag.

[8] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[9] Anthony Hall. Using z as a specification calculus for object-oriented systems. In H. Langmaack D. Bjorner, C. A. R. Hoare, editor, *VDM '90, VDM and Z - Formal Methods in Software Development*, volume 428, pages 290–318. Springer-Verlag, 1990.

[10] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[11] Ian Hayes. A generalisation of bags in z. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 113–127. Springer-Verlag, 1990.

[12] Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International, second edition, 1993.

[13] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. Clp(r) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference*, Logic Programming, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press.

[14] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. *The CLP(R) Programmer's Manual Version 1.2*, September 1992.

[15] M. Mowbray I. J. Hayes and G. A. Rose. Signalling system no. 7 the network layer. In G. Scollo E. Brinksma and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 3–14. Elsevier Science Publishers B. V. (North-Holland), 1990.

[16] K P Ishaq J J Masterson and A T Hockley. An approach to providing support tools for formal specification. In K. J. Turner, editor, *Formal Description Techniques*, pages 1–14. Elsevier Science Publishers B.V. (North-Holland), 1989.

[17] J-L Lassez J. Jaffar and M. J. Maher. *A Logic Programming Language Scheme*. Logic Programming: Relations, Functions and Equations. Prentice Hall, 1986.

[18] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM.

[19] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a clp system. In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference*, Logic Programming, pages 196–218, Melbourne, Australia, November 1986. MIT Press.

[20] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) language and system. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 14, pages 339–395, 1992.

[21] Paul W. King. Formalization of protocol engineering concepts. *IEEE Transactions On Computers*, 40(4):387–403, April 1991.

[22] R. D. Knott, P. J. Krause, and P. J. Byers. Animating set-theoretic specifications using prolog (collected papers). Technical report, University of Surrey, 1990.

[23] Ron Knott. A guide to using the "mathias" prolog library for discrete mathematics. Technical report, University of Surrey, March 1992.

[24] P.J. Krause and J. Cozens. Computer aided transformation of z into prolog. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 71–85. Springer-Verlag, 1990.

[25] Leslie Lamport. *LATEX , A Document Preparation System.* Addison-Wesley Publishing Company, 1986.

[26] Deyi Li and Dongbo Liu. *A Fuzzy Prolog Database system.* Research Studies Press Ltd, 1990.

[27] Jan van Katwijk Nico Plat and Hans Toetenel. Application and benefits of formal methods in softwrae development. *Software Engineering Journal*, pages 335–346, September 1992.

[28] Graeme Smith Paul King. Formalisation of behavioural and structural concepts for communication systems. In R. L. Probert L. Logrippo and H. Ural, editors, *Protocol Specification, Testing and Verification, X*, pages 3–18. Elsevier Science Publishers B. V. (North-Holland), 1990.

[29] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall International, 1991.

[30] C. Rees. Experiences using formal methods within the tmst project. *GEC Journal of Research*, 10(1):11–18, 1992.

[31] J. B. Roberts. *The Real Number System in an Algebraic Setting.* W. H. Freeman and Company, 1962.

[32] Anthony Lee Roger Duke, Gordon Rose. Object-oriented protocol specification. In R. L. Probert L. Logrippo and H. Ural, editors, *Protocol Specification, Testing and Verification, X*, pages 325–338. Elsevier Science Publishers B. V. (North-Holland), 1990.

[33] Paul King Gordon Rose Roger Duke, Ian Hayes. Protocol specification and verification using z. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 33–46. Elsevier Science Publishers B. V. (North-Holland), 1988.

[34] Einar Snekkenes. Authentication in open systems. In R. L. Probert L. Logrippo and H. Ural, editors, *Protocol Specification, Testing and Verification, X*, pages 311–324. Elsevier Science Publishers B. V. (North-Holland), 1990.

[35] J. M. Spivey. *The Z Notation A Reference Manual Second edition.* Prentice Hall International, 1992.

[36] Mike Spivey. *The Fuzz Manual.* Computing Science Consultancy, 2 Willow Close Garsington Oxford OX44 9 AN, second edition, July 1992.

[37] Rosalind Barden Susan Stepney and David Cooper. A survey of object orientation in z. *Software Engineering Journal*, pages 150–160, March 1992.

[38] C. S. Mellish W. F. Clocksin. *Programming in Prolog.* Springer-Verlag, 1984.

[39] Margaret M. West and Barry M. Eaglestone. Software development: two approaches to animation of z specifications using prolog. *Software Engineering Journal*, pages 264–276, July 1992.