International Journal of Engineering & Technology, 3 (1) (2014) 63-69
©Science Publishing Corporation
www.sciencepubco.com/index.php/IJET
doi: 10.14419/ijet.v3i1.1598
Research Paper

Z notation: A roadmap

Monika Singh [1]*, A .K. Sharma [2]

[1] *Faculty of Engineering and Technology (FET)*
[2] *Mody Institute of Technology and Science (MITS),Lakshmangarh, Rajasthan, India*
**Corresponding author E-mail: Dhariwal.monika@gmail.com*

Abstract

Formal methods offer the promise of significant improvements in verification and validation, and may be the only approach capable of demonstrating the absence of undesirable system behaviour. There are a number of formal specification languages for implementing Formal methods. This paper focus on Z-notation, a formal specification language and its semantics.

Keywords: *Formal methods, Formal specification language and Z- notation.*

1. Introduction

Formal methods used in developing the computer systems are mathematical based techniques foe describing system properties. A method is said to be formal if it has a sound mathematical basis, typically given by a formal specification language. The formal specification of a system is a set of mathematical expressions with well-defined syntax and semantic. By techniques of mathematical refinements, formal methods can be used at every stage of systems development in software life cycle. Due to their mathematical underpinning formal methods allow to specify systems with more precision, more consistency and less ambiguity. Formal methods of course do not guarantee correctness, but their use aims to increase our understanding of a system by revealing errors or aspects of incompleteness that might be expensive to rectify at a later date [1].
The use of formal methods can help to explore design choices. Such methods aid the design team in thinking about the operation of the system before its implementation. In particular, error conditions can be checked by calculating the precondition of an operation, and then dealing with the errors to ensure that the precondition of the complete operations true. When using informal methods, it is easy to gloss over such details until the implementation stage. The quality of Documentation of the system can be improved. By using the formal design as a basis for the manual for a system, it is likely that less information will be left out. Errors corrected at the design stage can be up to two orders of magnitude cheaper to correct than if they are found later and therefore the overall cost will be lowered.

2. Formal specification

There are a number of reasons why software construction is an inherently hard process to master. Specification plays a central role here; therefore, better means of specification improve productivity. One way of achieving this may be the use of formal specification languages, which have the advantage of being unambiguous. The primary idea behind a formal method is that there is benefit in writing a precise specification of a system, and formal methods use a formal or mathematical syntax for that purpose. This syntax is usually textual but can be graphical. Semantics is also provided, that is, a precise meaning is given for each description in the language. A specification of a system might cover one or more of a number of aspects, including its functional behaviour, its structure or architecture, or even cover aspects of non-functional behaviour such as timing or performance criteria. Formal specification is expression of traced attributes list, which is expressed by formal specification language and with specific abstraction level [2].

Formal specification languages can be further differentiated by the mean of different properties like process-oriented or sequential-oriented or it may be a model-oriented or a property-oriented, and what type of mathematics basis it uses in the following way:
Process-oriented .These is designed to describe concurrent networks of communicating component's behaviours, and since most of the languages describe system in term of process, so it is called as process-oriented.
Sequential-oriented – It provide the appropriate way to describe the input-output behavior of sequential system. But one problem with Sequential-oriented is that it lacks concurrence, synchronization and distribution.
Model-oriented .The language which fall in this category provide an abstract model of system and the operation is specified by either state change or by event affect the model.
Property-oriented. These focus on the property desired by the system being specified and on data type instead of services/ functions that the system provides.
Formal specification languages have better defined semantic. Based on these language attributes it is possible to conclude about most of represented knowledge including entire or partial code generation. Formal specification language creates mathematic basis for formal method. A variety of different formal specification techniques exist, some are general purpose while others stress aspects relevant to particular application domains (e.g., concurrent systems).Most are backed-up by varying degrees of tool support. Some of famous Languages are VDM, Z, B, OCL, SDL, and LOTOS. The definition of a formal semantics is based on a collection of data structures. We use the framework of mathematical domain theory [4] to define these data structures. Semantic domain is a fundamental concept of domain theory; it represents a set of elements that share some common properties. Also a semantic domain is accompanied by a set of operations as functions over the domain. A domain and its operations together form a semantic algebra [14].

3. Z notation

Z has been developed at Oxford University since the late 1970's by members of the Programming Research Group (PRG) within the Computing Laboratory. It is a typed language based on set theory and first order predicate logic.
The Z notation is a model oriented approach based on first order predicate logic [5] and Zermelo-Fraenkel (ZF) set theory [3] used for specifying behaviour of abstract data types and sequential programs. The notation was originated and inspired by Jean-Raymond Abrial while visiting the Oxford University Computing Laboratory, and was subsequently further developed by Hayes, Morgan, Sørensen, Spivey, Sufrin and others [6]. It is undergoing international standardization under ISO/IEC JTC1/SC22 [7]. Z is usually used for systems development because it describes state space of a system and operations which can be performed over it. Z notation is basically based on:

- First Order Predicate Logic
- Set theory and Relation

a) First-order logic
First-order logic is symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject. In first-order logic, a predicate can only refer to a single subject. First-order logic is also known as first-order predicate calculus or first-order functional calculus [7]. There are two logical constants in predicate logic, namely true and false. The basic elements of First Order Predicates are:

- Constants KingJohn, 2, NUS
- Predicates Brother, >,
- Functions Sqrt, LeftLegOf...
- Variables x, y, a, b...
- Connectives Ø, Þ, Ù, Ú, Û
- Equality =
- Quantifiers ", $

Connectives
The propositional connectives of negation, conjunction and disjunction are used

Table 1: Connectives

Notation	Name	Explanation
$\neg P$	negation	*true* iff P is *false*
$P \wedge Q$	conjunction	*true* iff P and Q are both *true*
$P \vee Q$	disjunction	*false* iff P and Q are both *false*

Quantifiers

- Allows statements about entire collections of objects rather than having to enumerate the objects by name.

- Universal quantifier- ∀x

Asserts that a sentence is true for all values of variable x

∀x Loves(x, FOPC)

∀x Whale(x) ⇒Mammal(x)

∀x Grackles(x) ⇒ Black(x)

∀x (∀y Dog(y) ⇒ Loves(x,y)) ⇒ (∀z Cat(z) ⇒ Hates(x,z))

•Existential quantifier- ∃

Asserts that a sentence is true for at least one value of a variable x

∃x Loves(x, FOPC)

∃x (Cat(x) ∧ Color(x, Black) ∧ Owns (Mary, x))

∃x (∀y Dog(y) ⇒ Loves(x, y)) ∧ (∀z Cat (z) ⇒Hates(x, z))

b) Set theory and Relation

Set

Set theory is a basis of modern mathematics, and notions of set theory are used in all formal descriptions. The notation used here is based on Zermelo-Fraenkel theory in which there are only sets. Members of sets can only be other sets. The word \element" may be used loosely when referring to set members treated as atomic, without regard to their set nature, and abstracting from their representation [8].

Basic set operations

Zermelo-Fraenkel set theory constructs its repertoire of set operations starting with the axiom of empty set, and then showing how to build up sets using the axioms of pairing and of union, and trimming back with the axiom of subset or separation**.**

Table 2: Set extensions and unions in metalanguage

Notation	Name	Explanation
$\varnothing$	empty set	$x \in \varnothing$ always false
$\{\}$	empty set	$= \varnothing$
$\{x\}$	singleton set	$y \in \{x\}$ iff $y = x$
$S \cup T$	union	the set of x such that $x \in S \vee x \in T$
$\{x, y, ..., z\}$	set extension	$= \{x\} \cup \{y, ..., z\}$
$\{i : S \mid P(i)\}$	comprehension	subset of elements i of S such that $P(i)$, by axiom of separation
$S \cap T$	intersection	$\{x : S \mid x \in T\}$
$S \setminus T$	difference	$\{x : S \mid x \notin T\}$

Powersets

The axiom of powers asserts the existence of powersets. The set of all finite subsets is a subset of this. The notation for these two forms is shown in Table3.

Table3: Powerset in metalanguage

Notation	Name	Explanation
$\mathbb{P}\, S$	set of all subsets	$T \in \mathbb{P}\, S$ iff $T \subseteq S$
$\mathbb{F}\, S$	set of all finite subsets	the smallest set containing the empty set and all singleton subsets of S and closed under the operation of forming the union of two sets

Numbers

Numbers are not primitive in Zermelo-Fraenkel set theory, but there are several well established ways of representing them. The choice of coding is not specified here. There are notations to measure the cardinality of a finite set, to define addition of natural numbers and to form the set of natural numbers between two stated natural numbers, as given in Table 4, where m and n stand for any expressions whose values are natural numbers:

Table 4: Operations on numbers in metalanguage

Notation	Explanation
$m + n$	sum of natural numbers m and n
$\# S$	cardinality of finite set S
$m \,..\, n$	set of natural numbers between m and n inclusive

Tuples and Cartesian products

In Z, tuples and Cartesian products may have two or more components. In this mathematical language, however, only pairs, which may be iterated, are used [9] [10].

Table 5: Tuples and Cartesian products in metalanguage

Notation	Explanation
(x, y)	pair
$x \mapsto y$	using "maplet", identical to (x, y)
$first\ p$	$first(x, y) = x$
$second\ p$	$second(x, y) = y$
$S \times T$	Cartesian product, set of ordered pairs whose first element is in S and whose second element is in T
$(x, y, ..., z)$	abbreviates $(x, (y, ..., z))$
$S \times T \times ... \times U$	abbreviates $S \times (T \times ... \times U)$

Relations

A relation is defined to be a set of Cartesian pairs. There are several operations involving relations, who are given equivalences in Table 5, where Q and Rare any relations, and S any set. A proposition about relations is given in Table 6:

Table 6: Relations in metalanguage

Notation	Name	Definition
$id\ S$	identity function	$\lambda\ x : S \bullet x$
$dom\ R$	domain	$\{p : R \bullet first\ p\}$
$R^{\sim}$	relational inversion	$\{p : R \bullet second\ p \mapsto first\ p\}$
$S \lhd R$	domain restriction	$\{p : R \mid first\ p \in S\}$
$S \lhd\!\!\!\!- R$	domain subtraction	$\{p : R \mid first\ p \notin S\}$
$R(\!\mid S \mid\!)$	relational image	$\{p : R \mid first\ p \in S \bullet second\ p\}$
$Q \fatsemi R$	relational composition	$\{q : Q; r : R \mid second\ q = first\ r \bullet first\ q \mapsto second\ r\}$
$Q \oplus R$	relational overriding	$((dom\ R) \lhd\!\!\!\!- Q) \cup R$

Functions

A function is identified with a particular form of relation, where each domain element has only one corresponding range element. The phrase \partial function" means exactly the same as the word \function" without qualification. Table 7 shows the various forms of function that are identified. In the table S and T are any sets, and the resultant expressions are sets of functions of various sorts.

Table 7: Functions in metalanguage

Notation	Name	Definition
$S \pfun T$	partial functions	$\{f : \mathbb{P}(S \times T) \mid \forall p, q : f \mid first\ p = first\ q \bullet second\ p = second\ q\}$
$S \fun T$	total functions	$\{f : S \pfun T \mid dom\ f = S\}$
$S \rightarrowtail T$	bijections	$\{f : S \fun T \mid f^{\sim} \in T \fun S\}$
$S \fin T$	finite functions	$\{f : \mathbb{F}(S \times T) \mid f \in S \pfun T\}$

4. Z schemas

A boxed notation called 'schemas' is used for structuring Z specifications. Schemas are primarily used to specify state spaces and operations for the mathematical modelling of systems. For example, here is a schema called StateSpace:

StateSpace

$x_1 : S_1; \ldots; x_n : S_n$

$Inv(x_1, \ldots, x_n)$

This schema specifies a state space in which x1, . . . , xn are the state variables and S1, . . . , Sn are expressions from which their types may be systematically derived. Inv (x1. . . xn) is the state invariant, relating the variables in some way for all possible allowed states of the system during its lifetime [11].
For example: The 'Birthday Book'- is a system for recording birthdays. It uses the following basic
Types (or given sets):
[NAME, DATE]
The state space of the Birthday Book is specified by:

$$\begin{array}{|l} \textit{BirthdayBook} \\ \hline known : \mathbb{P}\, NAME \\ birthday : NAME \pfun DATE \\ \hline known = \operatorname{dom} birthday \\ \hline \end{array}$$

The state variables are known (a number of people's names) and birthday (unique dates associated with each known person's name). The 'invariant' property of this schema is:
Known = dom birthday
i.e., every known person has a birth date associated with them. Z makes use of identifier decorations to encode intended interpretations. A state variable with no decoration represents the current (before) state and a state variable ending with a prime (0) represents the next (after) state. A variable ending with a question mark (?) represents an input and a variable ending with an exclamation mark (!) represents an output. A typical schema specifying a state change is the addition of a birthday to the existing Birthday book by the following operation schema:

$$\begin{array}{|l} \textit{AddBirthday} \\ \hline known : \mathbb{P}\, NAME \\ birthday : NAME \pfun DATE \\ known' : \mathbb{P}\, NAME \\ birthday' : NAME \pfun DATE \\ name? : NAME \\ date? : DATE \\ \hline name? \notin known \\ known = \operatorname{dom} birthday \\ known' = \operatorname{dom} birthday' \\ birthday' = birthday \cup \{name? \mapsto date?\} \\ \hline \end{array}$$

The entire state with its invariant is repeated for both the before (undashed) and after (dashed) states. The precondition of AddBirthday is name? /2 known. The operation part of the specification is the predicate
birthday0 = birthday [{name? 7! date?} which specifies that in the state after the operation AddBirthday is performed, the new value (birthday0) of the state variable birthday is birthday [{name? 7! date?}.We can also use one schema within another schema, which is called as schema inclusion. For example, AddBirthday can be specified using ΔBirthdayBook in the following way:

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

a) Schema operators
There are schema operators matching the logical connectives on predicates, such as ¬, ^, v, =>,⇔ and as well as quantification using $\forall$, $\exists$. The schemas are first normalized. For the binary connectives to be used there must be no conflicting
Declarations in the schemas being combined. For schema negation, the normalization is particularly important to ensure any hidden predicate constraints in the declarations are also negated. For binary operators, the declarations are merged in the resulting schema (remember that the order of the declarations is not important) and the predicate parts are combined depending on the operation involved. Z provides a 'pre' operator which may be used to return the precondition of a schema. Pre Operation existentially quantifies all after state and output components.
i.e.

$$\exists x_1' : S_1; \ldots; x_n' : S_n; o_1! : T_1; \ldots; o_p! : T_p \bullet Operation$$

b) Schema's Properties

An example of a simple property that one might want to prove is:

$$AddThenFindBirthday \vdash date! = date?$$

i.e., a FindBirthday operation after an AddBirthday, with the same name input to both, outputs the same date as that input to the AddBirthday operation. This is the sort of property that, if proved, provides an extra level of confidence that a specification is correct, since it confirms our intuitions about the properties of the specifications that we expect to hold. If a given desired property cannot be proved, this may well indicate a flaw in the specification, which can then be rectified at an early stage before any implementation has been started. Using informal design techniques, such errors are not normally discovered until a later stage, such as coding, testing, or even after the system has been delivered, with all the extra costs that this involves.

c) Tool Support

1) FuZZ - Z type-checker available commercially, together with an associated fuzz.sty LaTeX style file which has better fonts for the more esoteric Z symbols**.** The type-checker currently runs on Sun 3, Sun 4, IBM PC and VAX/VMS equipment.

2) CADiZ - provides support for Z using troff and LaTeX on UNIX system and Microsoft Word on PCs from York Software Engineering Ltd.

3) Cogito - methodology and toolset for the formal development of software. Uses the Sum specification and development language, based on the Z notation.

4) OOZE, an object-oriented Z environment.

5) PiZA, a Prolog Z animator.

6) JAPE (Just Another Proof Editor) by Bernard Sufrin and Richard Bornat supports typed set theory (Z).

7) Mathias Mathematics in Animation Suite, including advice on using it to animate Z.

8) OOZE, an object-oriented Z environment.

9) ZETA, an open environment based Z, providing an integration framework for tools to edit, browse, analyse, and animate Z specifications.

10) ZAST (Z Abstract Syntax Tree Viewer). Based on a PreCC Z grammar. Runs under Microsoft Windows 95 and Window NT.

11) Z/EVES, proving a Z front-end to the EVES proof tool based on ZF set theory.

12) ZTC is a Z type-checker available free of charge for educational and non-profit uses. It accepts LaTeX with zed or oz styles, and ZSL - an ASCII version of Z.

13) Zola, a commercial WYSIWYG editor, type-checker and tactical prover for Z from IST, UK.

14) ZANS is a Z animator. It is a research prototype that is still very crude. ZANS run on Linux, SunOS 4.x, Solaris 2.x, Windows 95 and NT 4.0.[13]

d) Advantages and Disadvantages of Z

Advantages

- The flexibility to model a specification which can directly lead to the code.
- Easy to understand
- A large class of structural models can be described in Z without higher order features, and can thus be analyzed efficiently.
- Independent Conditions can be added later

Disadvantages

Z is not ideal for all problems. For example, dealing with concurrency is clumsy. However, Z is good for systems which may be modelled as a sequence of operations on an abstract state. In general formal techniques require a significant amount of training effort and practical experience to be applied successfully. The toolset for Z is still not very advanced by industrial standards.

5. Conclusion

By making the use of formal specification language like Z notations we can reduce the error in the early phase of software development lifecycle.

References

[1] E.M. Clarke and J. Wing, "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, 1996.

[2] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo and D. Hamilton "Experiences Using Lightweight Formal Methods for Requirements Modeling," IEEE Transactions on Software Engineering , Vol. 24, No. 1, January, 1998

[3] J. M. Spivey, "Understanding Z: A Specification Language and its Formal Semantics, volume 3 of Cambridge Tracts in Theoretical Computer Science", Cambridge University Press, January 1988.

[4] S. Austin and G. I. Parkin, "Formal methods: A survey. Technical report", National Physical Laboratory, Queens Road, Teddington, Middlesex, TW11 0LW, UK, March 1993.
[5] M. Benveniste: Writing operational semantics in Z: A structural approach. In Prehn and Toetenel [338], pages 164–188.
[6] S. M. Brien, "The development of Z". In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, Semantics of Specification Languages (SoSL), Workshops in Computing, pages 1–14. Springer-Verlag, 1994
[7] S. M. Brien and J. E. Nicholls, " Z base standard", Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road Oxford, UK, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.
[8] M. Imperato, "An Introduction to Z", Chartwell-Bratt, 1991.
[9] D. Lightfoot, "Formal Specification using Z", Macmillan, 1991.
[10] J. M. Spivey, "Understanding Z: A Specification Language and its Formal Semantics, volume 3 of Cambridge Tracts in Theoretical Computer Science", Cambridge University Press, January 1988.
[11] J. M. Spivey, "An introduction to Z and formal specifications. IEE/BCS Software Engineering Journal, 4(1):40–50", January 1989.
[12] J. M. Spivey, "The Z Notation: a Reference Manual", Prentice Hall International Series in Computer Science, 2nd edition, 1992.
[13] www.cfdvs.iitb.ac.in/download/Docs/net/www.comlab.ox.ac.../z.html