Making Nondeterminism Explicit in Z

S.H. Mirian-HosseinAbadi[1] and M.R. Mousavi[2]

[1] Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
Email: mirian@sharif.ac.ir
[2] Department of Computer Science
Eindhoven University of Technology,
Eindhoven, The Netherlands
Email: m.r.mousavi@tue.nl

Abstract. Specification of system requirements is often involved with ambiguity and nondeterminism. Formal methods tend to mitigate ambiguity but nondeterminism remains as an inherent part of specification. This is due to the abstraction from real world details that causes a formal specification to define several results as a correct solution to a problem. Hence, a support for nondeterminism should be foreseen in formal methods.
In this paper, after studying different types of nondeterminism, some basic notations, namely multi- and power-schema, are added to Z formal language, to help explicit specification of nondeterminate constructs. Afterwards, a transformation is defined to generate nondeterministic semantics from specification in the same language. The results of adding the proposed notations is discussed from program development point of view.

Keywords: Nondeterminism, Z notation, Formal Specification, Program development.

1 Introduction

Using formal methods brings about precision in software development process but at the same time, the abstraction imposed in specification phase, introduces nondeterminism in formal specification. From program development point of view, having implicit nondeterminism is a disadvantage. Hence, to facilitate modelling nondeterminism in specifications special notations and semantics should be foreseen in formal specification languages.

In this paper we aim to present main ideas behind nondeterminacy and nondeterminism in set theoretic formal specifications, and add some notions of nondeterminacy to the Z specification language. To do this, in section 2, basic definitions and previous works on nondeterminism are reviewed. In section 3 nondeterministic constructs added to this language are introduced. Our approach is introduced, subsequently. The results achieved in our approach, is studied in section 4 from a program development point of view and an example is specified using the extended notation. Section 5 will summarize the results and show motivations for future work.

2 Nondeterminism in formal specification

2.1 Basic Concepts

The goal of formal specification is to define software requirements rigorously. This causes an abstraction from some unneeded details and allows different results in implementation. The high level abstraction introduces nondeterminism to the specification and causes the resulting artifact not be executable[4][1].

If we define software(software component) as a relation that computes output from an input set, nondeterministic software could produce different outputs from a single input[14, 10]. On the contrary, deterministic software computes the same result for each set of input. The syntactic notation for modelling nondeterministic semantics is called *nondeterminacy*[2]. Because of insufficient information from the system, some aspects of the system may remain undefined. These kinds of specification are called *under-specified.* There were some attempts to formalize the difference between the two concepts(nondeterminism and under-specification)[12].

Different types of nondeterministic constructs and their semantics were defined in the literature[10, 13]. Here, we restrict our attention to three major classifications:

1. Singular and plural nondeterminism: Singular and plural nondeterminism coincide with call-by-value and call-by-name semantics

[1] Although, some arguments were made in favor of (directly)executable specifications[3], this idea imposes extra restrictions on specifier(e.g. bounding variables' range) and the resulting program seems to be very inefficient.

[2] The problem wether a program(specification) using nondeterminate constructs acts nondeterministically, is shown to be undecidable[1].

in programming languages. In plural nondeterministic semantics, decisions regarding nondeterminism on a same structure are made independently. However, in singular semantics, always same decision is made in different places using same nondeterminate structure.

2. Bounded and unbounded choice: If the set of possible choices, could be an infinite set then the resulting nondeterminism is unbounded else it is called bounded nondeterminism. Fairness issues could be also discussed in unbounded nondeterministic semantics.
3. Nondeterminism and nontermination: Termination of an execution may depend on the choices made in nondeterministic construct. If a nondeterministic choice is made regardless of its impact on termination, it is called erratic. In contrast, if the choice is to be made in favor of termination(if at all possible) or nontermination, we should have angelic or demonic choice semantics, respectively.

2.2 Previous Works

Studying nondeterminism has a rich background in theoretical computer science. On the other hand before 80's there was not a concept of nondeterminism in mathematics. Hence, computer scientists used classic mathematical constructs to model nondeterminism. Recently, some works were done to develop nondeterministic mathematical framework on which formal models could be built[1].

A major branch of works on nondeterminism is done in the algebraic programs and specifications. A survey of works done in this field could be found in [13]. The basic methods introducing nondeterminism in algebraic specifications could be categorized as follows:

1. Functional models(operation level nondeterminism): In functional models, a nondeterministic operation is modelled by a set of deterministic functions. Hence, nondeterministic operations will consist of selecting an integer number between 1 and n(the number of deterministic functions) and applying the appropriate function. The main drawback of using this model is its intentional view over operations. In other words, two operations that have the same result may be distinguished in this model due to different function sets.

2. Multi-Algebra(result level nondeterminism): In this approach, a set-valued function will represent nondeterministic function that maps an input tuple to a set of outputs. In contrary with the former model, Multi-Algebra is focused on the behavior of the operations.
3. Power-Algebra(sort level nondeterminism): If the input sets are also promoted to a set of inputs(i.e. each argument is promoted to its power set) then a power-algebra is formed. This could be interpreted as if every possible outcome of applying the operation is associated with a member of the input set.
4. Relational models: In the relational models the restriction of using a defined function to model the operation is liberated and hence, the operation is modelled arbitrary subset of a defined relation.

In [5], Hussman proposed a nondeterministic term rewrite system. He first dealt with plural semantics and declared a soundness criteria in which plural semantics failed to be sound. The term rewrite rules are then restricted to expand nondeterministic terms with singular semantics.

Some works also concentrated on defining a logical framework for nondeterminism. Nondeterministic logical frameworks are defined as many valued(particularly three-valued) first order logics that admit variables to range over a set of value. Examples of nondeterministic three-valued logics are discussed in [9]. In [1], Blass and Gurevich defined a nondeterministic logic, called *Logic of Choice*. They introduced fixed and dependent choice operators(corresponding to singular and plural nondeterminism) to first order logic. The new choice operators consequently add nondeterministic terms(variable) to logic language. The idea of nondeterministic terms are applied here to model nondeterminism.

Also, Ward in his Ph.D. thesis[14] defined a nondeterministic refinement calculus. In his approach, he added specification constructs(set theoretic) to a functional language. Also, nondeterministic choice operators were added to specification language. Then, a set of refinement rules were proposed to support refining the specification to a nondeterministic functional language program. The resulting program uses *guess* construct to select a value nondeterministically from possible set of results. This kind of programming language construct was originally suggested by Floyd in [2], as a replacement for backtracking algorithms.

3 Our Approach

Nondeterministic expressions are defined in [14] as follows:

> An expression is nondeterministic if separate evaluations in the same state can give different results.

In this section, this definition will be formalized according to Z formal specification structure and then some tools are defined to support it in the same formal specification language.

3.1 Nondeterminism in Z

Z formal notation[11], is a formal specification language based on typed set theory and (classic) first order logic. In Z formal notation, specification constructs(e.g. axiomatic definitions and schemas) are used to modularize system's state and behavior. Among these constructs, schema is the most important tool to encapsulate specification chunks. Schema construct is used to model both system state(as state schema) and behavior(as operation schema).

A state schema encapsulate (a part of) system state variables with their invariants. An operation schema specifies a possible functionality or behavior on the system state by defining predicates that relate before-state variables(variables before application of the operation) and after-state ones. A valuation of variables in each schema is called its binding set. Most often an Init operation schema is defined on a state schema to define a special binding set as the schema initial state[3]. Then, each operation schema may map a pre-state to an after-state.

Hence, possibility of several after-state valuations for a single pre-state binding is a clear notion of nondeterminism in Z:

[3] General binding concept is shown in Z notation by $\langle\!\langle\, variable - name \leadsto variable - value \,\rangle\!\rangle$.

> An operation schema is nondeterministic if and only if there exists a single set of pre-state variable values that with two or more different sets of after-state variable values could satisfy schema predicates.

To formalize this concept, if an operation schema has the form $\mathit{Operation} \mathrel{\widehat{=}} [d \mid v]$ and, v_1 and v_2 are two valuations of *Operation* variables, then *Operation* is nondeterministic if and only if:

$$\begin{array}{l} \exists\, v_1, v_2 : \mathit{Operation}^v \bullet \\ \quad d_1^b = d_2^b \wedge \\ \quad \mathit{Operation}[v_1/\theta\,\mathit{Operation}] \wedge \mathit{Operation}[v_2/\theta\,\mathit{Operation}] \wedge \\ \quad v_1 \neq v_2 \end{array}$$

This definition asserts that the above two valuations agree on their pre-state and satisfy schema conditions but they are different only in their after-state. In the above predicate, d^b is used to denote before-state variables of schema and $\mathit{Operation}[v_1/\theta\,\mathit{Operation}]$ defines substitution of predicate variables in *Operation* with values in v_1 binding.

3.2 Multi-Schema

Multi-Schema is defined as a tool for specifier to specify nondeterminism explicitly. The concept is defined as follows:

> A Multi-Schema is a version of an operation schema with nondeterministic after-state variables.

By nondeterministic variable, we mean a variable that could contain more than a value in each state. This definition will help making nondeterminism in operation schemas explicit by providing set of possible after-states in their declaration part. But to define Multi-Schema formally there is a problem associated with schemas containing more than one after-state variable; The problem is that if we promote each after-state variable to its power type and place a universal quantifiers in front of the schema predicates to contain all possible values, the relationship between valuations that make the schema predicates true will disappear. In other words, resulting schema would loose its completeness and contain only a subset of after-state bindings for each variable that makes the predicates true with all bindings of all other variables. To overcome this problem, two solutions could be proposed:

- To force specifier combines all previous after-state variables in a new variable using cartesian product of their types and then apply the nondeterminism operator for Multi-Schema. This approach will solve the problem but will also impose restrictions on specifier, introduce major changes in schema predicate part, and make the resulting schema less legible.
- To encapsulate after-state variables in a schema and use a variable of its power type instead of all after-state declarations. This approach does not have the shortcomings of the first one.

Hence, we encapsulate after-sate variable of $Operation \mathrel{\widehat=} [declarations \mid predicated]$ in a new schema named $Operation^I$ (for Operation Interface):

$$Oparation^I \mathrel{\widehat=} [declarations^a]$$

and replace a fresh variable(named *interface*) of its power-type with previous after-state declarations in our nondeterministic transformation ($[\![Operation]\!]^\mu$):

$$\begin{aligned}
&MultiSchema = [\![Schema]\!]^\mu \mathrel{\widehat=}\\
&[interface : \mathbb{P}\, Operation^I;\ declaration^b \mid\\
&\quad \forall\, bind : Operation^I \bullet bind \in interface \Leftrightarrow\\
&\qquad predicates[forall\ v\ \in\ declaration^a :\ \ bind.(v^n)/(v^n))]
\end{aligned}$$

In the above definition $declaration^n$ function extracts variable name from their declaration($(variable : type)$).

Soundness and Completeness The definition of Multi-Schema is presented to be sound and complete in the following sense:

1. **Soundness:** A set of after-state bindings(for a particular pre-state) is present in Multi-Schema *interface* if it could satisfy original schema predicates.
2. **Completeness:** All set of after-state bindings(for a particular pre-state) that satisfy schema predicates are included in Multi-Schema *interface*.

The proof of both propositions is obvious. The right implication in Multi-Schema predicate($\forall\, bind : Operation^{I} \bullet bind \in interface \Rightarrow predicates$) provides soundness and left implication guaranties completeness.

An Example As a typical example of nondeterminism, the eight(N) queens problem is specified in this section. The problem is to place a number of queens($MaxQueens$) on a chess board such that none of them is able to capture another. The chess board is a square with $MaxDim$ number of ? in each row and column:

$$
\begin{array}{|l}
MaxDim : \mathbb{N} \\
MaxQeens : \mathbb{N}
\end{array}
$$

$NQueens$

$$
\begin{array}{|l}
setting : \mathrm{seq}(\mathbb{N} \times \mathbb{N}) \\
\hline
\forall\, x, y : \mathbb{N} \mid x \neq y \land x \in \mathrm{dom}\, setting \land y \in \mathrm{dom}\, setting \bullet \\
\qquad setting\ x \neq setting\ y \\
\forall\, x : \mathbb{N} \mid x \in \mathrm{dom}\, setting \bullet \\
\qquad first\ setting\ x \leq MaxDim \land second\ setting\ x \leq MaxDim
\end{array}
$$

The $NQueens$ schema defines the chess board in which no two queens can be in one place and they are all in the board limits. Then $Init$ and $Arrange$ schemas will be defined to specify initial and solution states:

$NQueensInit$

$$
\begin{array}{|l}
\Delta NQueens \\
\hline
setting' = \varnothing
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Arrange}} \\
\Delta \mathit{NQueens} \\
\hline
\#\mathit{setting}' = \mathit{MaxQueens} \\
\forall\, x, y : \mathbb{N} \mid x \in \operatorname{dom} \mathit{setting}' \land y \in \operatorname{dom} \mathit{setting}' \bullet \\
\qquad \mathit{first}\ \mathit{setting}'\ x \neq \mathit{first}\ \mathit{setting}'\ y \land \\
\qquad \mathit{second}\ \mathit{setting}'\ x \neq \mathit{second}\ \mathit{setting}'\ y \land \\
\qquad \mathit{second}\ \mathit{setting}'\ x + \mathit{first}\ \mathit{setting}'\ x \neq \mathit{second}\ \mathit{setting}'\ y + \mathit{first}\ \mathit{setting}'\ y \\
\qquad \mathit{second}\ \mathit{setting}'\ x - \mathit{first}\ \mathit{setting}'\ x \neq \mathit{second}\ \mathit{setting}'\ y - \mathit{first}\ \mathit{setting}'\ y \\
\hline
\end{array}
$$

But if one generates a program that simulates the above schema(particularly through proof methods) there is no guarantee that he could be able to get all the solutions of the problem, nor he will be sure that he could see a particular solution at all. Hence, we apply nondeterministic transformation on *Arrange* schema:

$$\mathit{MultiArrange} \mathrel{\widehat{=}} [\![\mathit{Arrange}]\!]^{\mu}$$

Using this transformation, the generated program from *MultiArrange* schema will assure generation of all possible outcomes:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Arrange}^{I}} \\
\mathit{setting}' : \operatorname{seq}(\mathbb{N} \times \mathbb{N}) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{MultiArrange} \\
\hline
\textit{setting} : \mathrm{seq}(\mathbb{N} \times \mathbb{N}) \\
\textit{interface} : \mathbb{P}\, \textit{Arrange}^{I} \\
\hline
\forall\, \textit{bind} : \textit{Arrange}^{I} \bullet \textit{bind} \in \textit{interface} \Leftrightarrow \\
\quad \# \textit{bind.setting}' = \textit{MaxQueens} \\
\quad \forall\, x, y : \mathbb{N} \mid x \in \mathrm{dom}\, \textit{bind.setting}' \land y \in \mathrm{dom}\, \textit{bind.setting}' \bullet \\
\qquad \textit{first bind.setting}'\ x \neq \textit{first bind.setting}'\ y \land \\
\qquad \textit{second bind.setting}'\ x \neq \textit{second bind.setting}'\ y \land \\
\qquad \textit{second bind.setting}'\ x + \textit{first bind.setting}'\ x \neq \\
\qquad\quad \textit{second bind.setting}'\ y + \textit{first bind.setting}'\ y \\
\qquad \textit{second bind.setting}'\ x - \textit{first bind.setting}'\ x \neq \\
\qquad\quad \textit{second bind.setting}'\ y - \textit{first bind.setting}'\ y \\
\quad \forall\, x, y : \mathbb{N} \mid x \neq y \land x \in \mathrm{dom}\, \textit{setting} \land y \in \mathrm{dom}\, \textit{setting} \bullet \\
\qquad \textit{setting}\ x \neq \textit{setting}\ y \\
\quad \forall\, x : \mathbb{N} \mid x \in \mathrm{dom}\, \textit{setting} \bullet \\
\qquad \textit{first setting}\ x \leq \textit{MaxDim} \land \textit{second setting}\ x \leq \textit{MaxDim} \\
\quad \forall\, x, y : \mathbb{N} \mid x \neq y \land x \in \mathrm{dom}\, \textit{bind.setting}' \land y \in \mathrm{dom}\, \textit{bind.setting}' \bullet \\
\qquad \textit{bind.setting}'\ x \neq \textit{bind.setting}'\ y \\
\quad \forall\, x : \mathbb{N} \mid x \in \mathrm{dom}\, \textit{bind.setting}' \bullet \\
\qquad \textit{first bind.setting}'\ x \leq \textit{MaxDim} \land \\
\qquad \textit{second bind.setting}'\ x \leq \textit{MaxDim} \\
\hline
\end{array}
$$

3.3 Schema calculus operators

As the Multi-Schema transformation changes schema declarations, schema calculus operators will no longer work properly on Multi-Schemas. Hence, simple logical operators of schema calculus are re-defined to apply semantics of conjunction, disjunction, implication and negation operators and then apply nondeterministic transformation:

$$
\begin{array}{l}
[\![\textit{Schema}_1]\!]^{\mu} \bigcirc\hspace{-0.95em}\Box\ [\![\textit{Schema}_2]\!]^{\mu} = \\
[\![[\textit{declaration}_1 \mid \textit{predicates}_1]]\!]^{\mu} \bigcirc\hspace{-0.95em}\Box \\
\qquad [\![[\textit{declaration}_2 \mid \textit{predicates}_2]]\!]^{\mu} == \\
[\![[\textit{declaration}_1 \cup \textit{declaration}_2 \mid \textit{predicates} \Box \textit{predicates}_2]]\!]^{\mu} \\
\ \textit{where}\ \Box \in \{\lor, \land, \Rightarrow\}
\end{array}
$$

$$
\begin{aligned}
&\neg[\![Schema_1]\!]^{\mu} = \\
&\neg[\![[declaration_1 \mid predicates_1]]\!]^{\mu} == \\
&[\![[declaration_1 \mid \neg predicates_1]]\!]^{\mu}
\end{aligned}
$$

Also, to define new semantics for sequential composition operator on Multi-Schema. We have to change the second schema declarations so that it explicitly admits multiple bindings of pre-state variable. This idea leads to a new kind of nondeterministic schemas named here *Power-Schema* and defined with:

$$Operation^{P} \mathrel{\widehat{=}} [declaration^{b}]$$

$$
\begin{aligned}
&PowerSchema = [\![Schema^{\rho}]\!] \mathrel{\widehat{=}} \\
&[preState : \mathbb{P}\, Operation^{p};\ interface : \mathbb{P}\, Operation^{I} \mid \\
&\quad \forall\, bind_i : Operation^{I} \bullet \exists\, bind_p : Operation^{p} \bullet \\
&\qquad bind_i \in interface \Leftrightarrow \\
&\qquad\quad declarations(forall\ v\ in\ v^{a}\ bind_i.v/v;\ forall\ v\ in\ v^{b} \bullet bind_p.v/v)
\end{aligned}
$$

Now, new sequential composition operator($⨾$) is defined with:

$$
\begin{aligned}
&[\![Schema_1]\!]^{\mu} ⨾ [\![Schema_2]\!]^{\mu} = \\
&[\![[declaration_1 \mid predicates_1]]\!]^{\mu} ⨾ [\![[declaration_2 \mid predicates_2]]\!]^{\mu} \mathrel{\widehat{=}} \\
&[\![[declaration_1 \mid predicates_1]]\!]^{\mu} \wedge [\![[declaration_2 \mid predicates_2]]\!]^{\rho} \wedge \\
&\quad [interface : Schema_1^{I};\ preState : Schema_2^{P} \mid \\
&\quad forall\ v \in declaration_1^{a} \cap declaration_2^{b} : \\
&\qquad interface.v = preState.v]
\end{aligned}
$$

4 Nondeterminism and program development

Having specified a problem formally, the question arises as to how the program can be developed so as to guarantee that it meets the requirements. If the formal specification languages have a weakness that prevents their wider industrial use, it could be their inability to bridge the gap from specification to (verifiably correct) programs. Several approaches are taken into account for developing programs from formal specifications. These approaches can be categorized as *refinement*, *animation* and *extraction*[7]. The refinement approach makes changes to a specification to produce an executable code in an stepwise manner[8]. All the activities which concern developing programs directly from specifications by applying a set of *rules of thumb* can be classified as *animation*. Research in this area has ranged from the simple animation of specifications to the design of new programming languages[7]. The extraction approach consists of two major phases, namely, a) functional specification and b) proof of existence. Programs can be extracted automatically given the output of the latter stage. The advantage of this approach is this fact that the only proof which is needed is the proof of correctness, and the rest is automatic[7].

A formal specification can contain nondeterministic components which make the process of program development in either of the above mentioned paradigms an uneasy task. Our approach can be used to facilitate this process. In other words, because we generate the nondeterministic semantics of a specification in the same formal language, a normal program extraction process will result in a program with all possible nondeterministic choices.

5 Conclusion and Future Works

In this paper, we introduced a some notations and their semantics for defining nondeterminism in Z explicitly. These notation, followed by transformation rules extends the possible choices of nondeterminism to deterministic operation schemas. Presented nondeterministic transformation could be considered as an unbounded, plural semantics for nondeterminism in Z. Using these constructs in a program development process will lead to a program that uncovers different possible valuation of specification schemas.

The proposed constructs could be trivially used in defining angelic and demonic choice semantics in the same language[6]. Also, a choice probability could be associated to participating schemas to develop a probabilistic schema calculus. Another interesting topic in continuing this research could be studying extension of choice constructs to resulting program using nondeterministic functional languages.

References

1. Blass A., and Gurevich Y., The Logic of Choice, *Journal of Symbolic Logic*, 65(3):1264–1310, 2000.
2. Robert W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14:636–644, 1967.
3. Fuchs N., Specifications are (preferably) executable. *IEE Software Engineering Journal*, 7(5):323–334, September 1992.
4. Hayes I. and Jones, C., Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338, November 1989.
5. Hussman H., *Nondeterminism in Algebraic Specifications and Programs*, Birkhauser, 1993.
6. Mousavi M.R., Nondeterminism in Set-Theoretic Formal Specification: A Constructive Approach(in Farsi), M.Sc. Thesis, Department of Computer Engineering, Sharif University of Technology, 2001.
7. Mirian-Hosseinabadi S.-H., Constructive Z, Ph.D. Thesis, Department of Computer Science, University of Essex, 1997.
8. Morgan C., Programming from Specifications, Second Edition, Prentice Hall, 1994.
9. Pappinghaus P., and Wirsing M., Nondeterministic Three-Valued Logic: Isotonic and Guarded Truth-Functions, *Studia Logica*, XLII.1 1–22, 1983.
10. Söndergaard H., and Sestoft P., Non-Determinism in Functional Languages, *The Computer Journal*, 35(5):514–523, October 1992.
11. Spivey, J.M., *The Z notation.* Prentice-Hall, 1989.
12. Walicki M., and Broy M., Structured specifications and implementation of nondeterministic data types. *Nordic Journal of Computing*, 2(3):358–395, Fall 1995.
13. Walicki M., and Meldal S., Algebraic approches to nondeterminism: An overview. *ACM Computing Surveys*, 29(1):30–81, March 1997.
14. Ward N.T.E., *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, University of Queensland, Australia, 1994.