

# The Z specification language

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering  
Concordia University Montreal, Canada

February 1, 2022

# Introduction

- ▶ Z (pronounced “zed”) is a *specification language* based on predicate logic and set theory.
- ▶ A formal specification in Z describes *what* the system does, without saying *how* it does it.
- ▶ The formal specification is implementation independent.

## Example: Birthday book

- ▶ In this example we are defining the specification of a birthday book.
- ▶ We introduce the set of all names, and the set of all dates, as basic types in the specification:

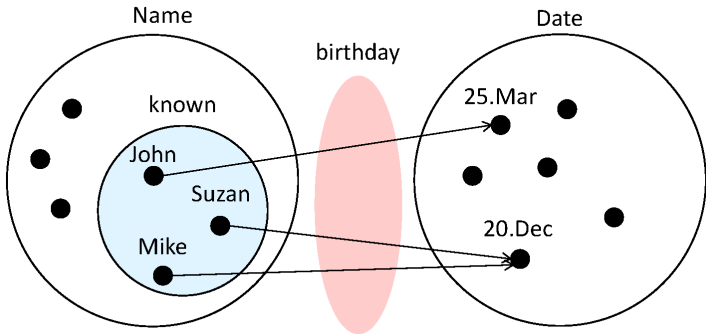
$[Name, Date]$

- ▶ One possible state of the system has three people in the set *known*, with their birthdays recorded by the function *birthday*:

$known = \{John, Mike, Suzan\}$

$birthday =$   
 $\{$   
     $John \mapsto 25.Mar,$   
     $Mike \mapsto 20.Dec,$   
     $Suzan \mapsto 20.Dec$   
 $\}$

## Birthday book /cont. - Function *birthday*



## Birthday book: State schema

- ▶ A specification is decomposed into *schemata* (plural for *schema*).
- ▶ A *Z schema* can provide a high-level description of a model.

<i>BirthdayBook</i>	_____
<i>known</i> : $\mathbb{P}$ <i>Name</i>	
<i>birthday</i> : <i>Name</i> $\rightarrow$ <i>Date</i>	
<i>known</i> = dom <i>birthday</i>	

- ▶ Variable *known* is the set of names with birthdates recorded and *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.
- ▶ Note that *birthday* is a partial function: It does not relate every element of *Name* to *Date*.

## Schemata /cont.

- ▶ We define a schema for both the data of the model as well as for each of the system's operations.

<i>Name</i> _____
<i>Declaration part</i> ( <i>state</i> )
<i>Predicate part</i>

- ▶ A schema consists of
- ▶ **Name**
- ▶ **Declaration part** (**state**): specifies a set of variable assignments and their types (called the *state* of the system).
- ▶ **Predicate part**
  - ▶ Properties of the model expressed in terms of the variables defined in the declaration part.
  - ▶ These properties constitute *state invariants* (must be true at all times).

## Schemata /cont.

- ▶ Alternatively a schema can be expressed in a linear notation as

$$name \hat{=} [declaration\ part \mid predicate\ part]$$

## Decorations: Before and after state

- ▶ *Decorations* give identifiers conventional meanings.
- ▶ A variable decorated with the symbol ' denotes its value after having being manipulated by an operation.
- ▶ Thus, *known* and *birthday* denote the states of their corresponding sets before an operation is invoked, whereas *known'* and *birthday'* denote the states of their corresponding sets after an operation has been successfully terminated.



## A birthday book /cont.

- ▶ The predicate  $known = \text{dom } birthday$  states that the set  $known$  is the same as the domain of the function  $birthday$ , i.e. the set of names that the function can apply.
- ▶ This relationship is an *invariant* of the system.

*BirthdayBook*

$known : \mathbb{P} \text{ Name}$

$birthday : \text{Name} \rightarrow \text{Date}$

$known = \text{dom } birthday$

## A birthday book: State schema

- ▶ In our schema we have managed to capture the information that
  - 1 Each person can have only one birthday (because *birthday* is a function), and
  - 2 Two (or more) people can share the same birthday (as in the example).
- ▶ The invariant  $known = \text{dom } birthday$  is satisfied, because *birthday* records a date for exactly the three names in set *known*.

## A birthday book: Operation *AddBirthday*

- ▶ Once the state space is defined, we must provide a schema for each of the operations of the system.
- ▶ Operation *AddBirthday* adds a new birthday into the system:

<i>AddBirthday</i>	_____
$\Delta BirthdayBook$	
$name? : Name$	
$date? : Date$	
_____	
$name? \notin known$	
$known' = known \cup \{name?\}$	
<u><math>birthday' = birthday \cup \{name? \mapsto date?\}</math></u>	

- ▶ The declaration  $\Delta BirthdayBook$  captures the fact that the schema describes a *state change*: it introduces variables  $known$ ,  $birthday$ , and  $known'$ ,  $birthday'$ .
- ▶ The first is an observation of the state before the change, and the last two are observations of the state after the change.

## A birthday book: Operation *AddBirthday*

- ▶ Next come the declarations of the two inputs to the operation and by convention, the names of inputs end in a question mark.

*name?* : *Name*

*date?* : *Date*

## A birthday book: Operation *AddBirthday*

- ▶ The first line under the dividing line defines the precondition to the operation:

*name?  $\notin$  known*

- ▶ The name to be added must not already be one of those known to the system, as each person can have only one birthday.
- ▶ The specification can be extended to include error conditions in the case of a precondition failure.

## A birthday book: Operation *AddBirthday*

- ▶ If the precondition is satisfied, the next two lines specify the postcondition to the operation:

$$\begin{aligned} \textit{known}' &= \textit{known} \cup \{\textit{name}?\} \\ \textit{birthday}' &= \textit{birthday} \cup \{\textit{name}? \mapsto \textit{date}?\} \end{aligned}$$

- 1 *name?* is now a member of the set *known*.
- 2 The birthday function is extended to map the new name to the given date.

## A birthday book: Operation *FindBirthday*

- ▶ Operation *FindBirthday* finds the birthday of a person known to the system:

<i>FindBirthday</i>	_____
$\exists \text{BirthdayBook}$	
$name? : \text{Name}$	
$date! : \text{Date}$	
<hr/>	
$name? \in \text{known}$	
$date! = \text{birthday}(name?)$	

- ▶ The declaration  $\exists \text{BirthdayBook}$  indicates that this is an operation in which the state of the system does not change.
- ▶ It makes the following statements redundant:

$$\begin{aligned} \text{known}' &= \text{known} \\ \text{birthday}' &= \text{birthday} \end{aligned}$$

## Operation *FindBirthday*: Declaring an output

- ▶ The use of a name ending in an exclamation mark defines an output:

*date!* : *Date*



## Operation *FindBirthday*: Precondition and output

- ▶ The precondition of the operation is that *name?* is one of the names known to the system:

$$name? \in known$$

- ▶ If this is so, then the output *date!* is the value of the birthday function given argument *name?*.

$$date! = birthday(name?)$$

## A birthday book: Operation *Remind*

- ▶ Operation *Remind* finds which people have a birthday on a given date:

<i>Remind</i>	_____
$\exists \text{BirthdayBook}$	
<i>today?</i> : <i>Date</i>	
<i>cards!</i> : $\mathbb{P}$ <i>Name</i>	
<hr/>	
$\text{cards!} = \{n : \text{known} \mid \text{birthday}(n) = \text{today?}\}$	

- ▶ This time there is no precondition.
- ▶ The output *cards!* is defined as the set of all values drawn from the set *known* such that the value of the birthday function at *n* is *today?*.

## Initializing the system

- ▶ To complete the specification, we must state the initial state of the system:

<i>InitBirthdayBook</i>	_____
<i>BirthdayBook</i>	
<i>known</i> = $\emptyset$	

## Handling errors

- ▶ What will happen if the user tries to add a birthday for someone already known to the system? Or if they try to find the birthday of someone not known?
- ▶ Our current specification makes no claim about what should happen: The system may ignore incorrect input, or it may break down (among other things).
- ▶ In addition to the basic specification, we can describe, separately the errors which might be detected and the desired responses to them.
- ▶ We can then use operations to combine the two descriptions into a stronger (robust) specification.

## Handling errors /cont.

- ▶ We define Report as an enumerated set of values that an output variable *result!* may assume:

$$Report ::= ok \mid already\_known \mid not\_known.$$

- ▶ We can now define a schema that specifies that the result should be *ok*.

$$\frac{Success \quad \exists BirthdayBook \quad result! : Report}{result! = ok}$$

- ▶ We can now combine this description with *AddBirthday* as

$$AddBirthday \wedge Success$$

## Handling errors /cont.

- ▶ For each precondition error we must define a schema which describes the conditions under which the error occurs and specifies the appropriate report to be produced.
- ▶ Schema *AlreadyKnown* specifies that the report *already\_known* should be produced when the input *name?* is already a member of *known*:

*AlreadyKnown* \_\_\_\_\_

$\exists$  *BirthdayBook*

*name?* : *Name*

*result!* : *Report*

*name?*  $\in$  *known*

*result!* = *already\_known*

## Strengthening the specification: *RAddBirthday*

- ▶ We can now provide a robust version of *AddBirthday*:

$$RAddBirthday \hat{=} (AddBirthday \wedge Success) \oplus AlreadyKnown$$

- ▶ This definition introduces a new schema obtained by combining the schemas on the right-hand side.
- ▶ Alternatively, operation *RAddBirthday* could be specified directly by writing a single schema which combines the predicate parts of the three schemas *AddBirthday*, *Success* and *AlreadyKnown*.

## Strengthening the specification: *RFindBirthday*

- ▶ A robust version of operation *FindBirthday* must be able to report if the input name is not known:

<i>NotKnown</i>
$\exists \text{BirthdayBook}$
$\text{name?} : \text{Name}$
$\text{result!} : \text{Report}$
$\text{name?} \notin \text{known}$
$\text{result!} = \text{not\_known}$

- ▶ The robust operation either behaves as described by *FindBirthday* and reports success, or reports that the name was not known:

$$RFindBirthday \hat{=} (FindBirthday \wedge Success) \oplus NotKnown$$



## Strengthening the specification: *RRemind*

- ▶ Operation *Remind* never results in an error, so its robust version need only add the reporting of success:

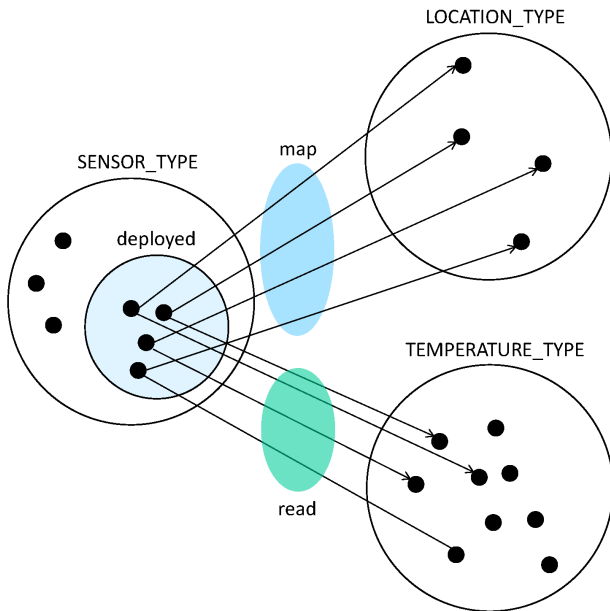
$$RRemind \hat{=} Remind \wedge Success$$

## Example: Temperature monitoring

- ▶ A system maintains a number of sensors, where each is deployed in a separate location in order to read the location's temperature.
- ▶ Before the system is deployed, all locations are marked on a map, and each location will be addressed by a sensor.
- ▶ The formal specification of the system introduces the following three types:

*SENSOR\_TYPE,*  
*LOCATION\_TYPE,*  
*TEMPERATURE\_TYPE*

# Visualization of the system



## State schema

*TempMonitor*

*deployed* :  $\mathbb{P} \text{SENSOR\_TYPE}$

*map* :  $\text{SENSOR\_TYPE} \rightarrow \text{LOCATION\_TYPE}$

*read* :  $\text{SENSOR\_TYPE} \rightarrow \text{TEMPERATURE\_TYPE}$

*deployed* = dom *map*

*deployed* = dom *read*

## Operation DeploySensorOK

- ▶ Operation DeploySensorOK places a new sensor to a unique location.
- ▶ We may assume that some (default) temperature is also passed as an argument.

*DeploySensorOK* \_\_\_\_\_

$\Delta$  *TempMonitor*

*sensor?* : *SENSOR\_TYPE*

*location?* : *LOCATION\_TYPE*

*temperature?* : *TEMPERATURE\_TYPE*

*sensor?*  $\notin$  *deployed*

*location?*  $\notin$  *ran map*

*deployed'* = *deployed*  $\cup$  {*sensor?*}

*map'* = *map*  $\cup$  {*sensor?*  $\mapsto$  *location?*}

*read'* = *read*  $\cup$  {*sensor?*  $\mapsto$  *temperature?*}

# Operation ReadTemperatureOK

- ▶ Operation ReadTemperatureOK obtains the temperature reading from a sensor, given the sensor's location.

*ReadTemperatureOK* \_\_\_\_\_

$\exists$  *TempMonitor*

*location?* : *LOCATION\_TYPE*

*temperature!* : *TEMPERATURE\_TYPE*

*location?*  $\in \text{ran } \textit{map}$

*temperature!* = *read*( $\textit{map}^{-1}(\textit{location?})$ )

## Success and error schemata

- ▶ We introduce an enumerated type *MESSAGE* which will assume values that correspond to success and error messages.

*Success*

$\exists TempMonitor$

*response!* : *MESSAGE*

*response!* = 'ok'

## Error schema SensorAlreadyDeployed for DeploySensorOK

- ▶ Operation DeploySensorOK places the following precondition:  
*sensor?*  $\notin$  *deployed*
- ▶ We provide an error schema for the case where this precondition fails:

*SensorAlreadyDeployed* \_\_\_\_\_

$\exists$  *TempMonitor*

*sensor?* : *SENSOR\_TYPE*

*response!* : *MESSAGE*

*sensor?*  $\in$  *deployed*

*response!* = '*Sensor deployed*'



## Error schema LocationAlreadyCovered for DeploySensorOK

- ▶ Operation DeploySensorOK places the following precondition:  
 $location? \notin \text{ran } map$
- ▶ We provide an error schema for the case where this precondition fails:

*LocationAlreadyCovered* \_\_\_\_\_

$\exists TempMonitor$

$location? : LOCATION\_TYPE$

$response! : MESSAGE$

$location? \in \text{ran } map$

$response! = 'Location\ already\ covered'$

## Error schema LocationUnknown for ReadTemperatureOK

- ▶ Operation ReadTemperatureOK places the following precondition:  $location? \in \text{ran } map$
- ▶ We provide an error schema for the case where this precondition fails:

*LocationUnknown*

$\exists TempMonitor$

$location? : LOCATION\_TYPE$

$response! : MESSAGE$

$location? \notin \text{ran } map$

$response! = 'Location not covered'$

## Defining reliable operations

$$\begin{aligned} \textit{DeploySensor} \hat{=} & \\ & (\textit{DeploySensorOK} \wedge \textit{Success}) \oplus \\ & (\textit{SensorAlreadyDeployed} \vee \textit{LocationAlreadyCovered}) \end{aligned}$$

$$\begin{aligned} \textit{ReadTemperature} \hat{=} & \\ & (\textit{ReadTemperatureOK} \wedge \textit{Success}) \oplus \textit{LocationUnknown} \end{aligned}$$

# Bibliography

1. V. S. Alagar and K. Periyasamy, *Specification of Software Systems*, 2nd. ed., Springer, 2011.
2. J. Jacky *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.
3. J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd. ed., Prentice Hall International (UK) Ltd, 1992.

## Appendix A: Operations on sets

Operator	Synopsis	Meaning
$\in$	$x \in S$	set membership
$\cup$	$S_1 \cup S_2$	set union
$\cap$	$S_1 \cap S_2$	set intersection
$\setminus$	$S_1 \setminus S_2$	set difference
$\#$	$\#S$	cardinality of a set
$\subseteq$	$S_1 \subseteq S_2$	subset
$\subset$	$S_1 \subset S_2$	proper subset
$=$	$S_1 = S_2$	set equality
$\bigcup$	$\bigcup SS$	generalized union of sets $SS$
$\bigcap$	$\bigcap SS$	generalized intersection of sets $SS$
$\mathbb{P}$	$\mathbb{P} S$	power set of the set $S$
$\mathbb{F}$	$\mathbb{F} S$	finite subsets of the set $S$

## Appendix B: Notations for functions

Symbol	Meaning
$\rightarrow$	Total function
$\dashrightarrow$	Partial function
$\rightharpoonup$	Total injective function
$\dashv\rightharpoonup$	Partial injective function
$\twoheadrightarrow$	Total surjective function
$\dashv\twoheadrightarrow$	Partial surjective function
$\rightharpoonup\twoheadrightarrow$	Partial bijective function
$\twoheadrightarrow$	Total bijective function
$\dashv\rightarrow$	Finite partial function
$\dashv\rightharpoonup$	Finite partial injective function

## Appendix C: Operations on relations and functions

Operator	Synopsis	Meaning
$\leftrightarrow$	$X \leftrightarrow Y$	declaration of a binary relation between X and Y
$\mapsto$	$x \mapsto y$	maplet
dom	dom R	domain of the relation R
ran	ran R	range of the relation R
id	id X	identity relation
$\circ$	$R_1 \circ R_2$	relational composition
$\circ$	$R_1 \circ R_2$	backward relational composition
$\triangleleft$	$S \triangleleft R$	domain restriction
$\triangleright$	$R \triangleright S$	range restriction
$\triangleleft$	$S \triangleleft R$	domain subtraction (domain anti-restriction)
$\triangleright$	$R \triangleright S$	range subtraction (range anti-restriction)
$\sim$	$R \sim$	relational inverse
$-(  -  )$	$R (  S  )$	relational image
$\oplus$	$R_1 \oplus R_2$	relational overriding
	$R^k$	relational iteration
	$R^+$	transitive closure of the relation R
	$R^*$	reflexive transitive closure of the relation R

## Appendix D: Operations on sequences

Operator	Synopsis	Meaning
#	# S	length of the sequence S
$\wedge$	$S_1 \wedge S_2$	concatenation of sequence $S_1$ with $S_2$
<i>rev</i>	<i>rev</i> S	reverse of the sequence S
<i>head</i>	<i>head</i> S	first element of the sequence S
<i>last</i>	<i>last</i> S	last element of the sequence S
<i>tail</i>	<i>tail</i> S	sequence S with its first element removed
<i>front</i>	<i>front</i> S	sequence S with its last element removed
$\wedge /$	$\wedge / SS$	distributed concatenation of the sequence of sequences SS
$\subseteq$	$S \subseteq T$	S is a sequence forming the prefix of the sequence T
<i>suffix</i>	S <i>suffix</i> T	S is a sequence forming the suffix of the sequence T
<i>in</i>	S <i>in</i> T	S is a segment inside the sequence T
$\upharpoonright$	$U \upharpoonright S$	extract the elements from the sequence S corresponding to the index set U; the result is also a sequence, maintaining the same order as in S
$\upharpoonright$	$S \upharpoonright V$	extract the elements of the set V from the sequence S; the result is also a sequence, maintaining the same order as in S
<i>disjoint</i>	<i>disjoint</i> SeqSet	SeqSet is an indexed family of mutually distinct sets
<i>partitions</i>	SeqSet <i>partitions</i> T	the indexed family of mutually disjoint sets whose distributed union is T