Daffodil International University

Department of Software Engineering

Documentation of Software Engineering

Z specification language

Presenter

Md Alamgir Kabir

sagar.whu@outlook.com

Outline

- Z notation

Daffodil International University

Department of Software Engineering

Motivation
Problem Statements
Contributions

INTRODUCTION

Example

A student may join the class if the class is not already full and if the student has not already enrolled. A new student cannot have passed all their assignments.

EnrolOk

$\Delta Class$

$student? : STUDENT$

$\#enrolled < maxClassSize$

$student? \notin enrolled$

$enrolled' = enrolled \cup \{ student? \}$

$passed' = passed$

The *?* in *student?* means input.

Zed Specification Language

- Based on typed set theory
- The most widely-used formal specification language
- Built upon **schemas**
  - Basic building blocks
  - Allow modularity
  - Easier to understand by using graphical presentation
- pronounced "Zed"

Set Display

{ tom, anne, jerry }
{ 1988, 1992, 1996, 2000, 2004 }
{ abc, bca, cab, acb, bac, cba }

- We use curly brackets to separate the list from its surroundings.
- The objects that make up a set are known as its *elements* or *members*.
- We separate one member from the next by a comma.

Naming a Set

- If a set has too many members to conveniently list, we can simply give them descriptive names.

PERSON - the set of all people
LEAPYEAR - the set of all leap years ever
PASSWORD - the set of all possible passwords

- We follow the convention that names we choose for our sets
  - are written entirely in capital letters - *PERSON* not Person
  - are singular - *PASSWORD* not PASSWORDS
  - do not contain spaces, underscores or hyphens - *LEAPYEAR* not LEAP YEAR

Some Standard Sets

- Some sets have names already assigned to them.
  - $\mathbb{Z}$ (say fat zed) - the set of all whole numbers, negative, zero and positive. -1, 0, 1 are members of this set.
  - $\mathbb{N}$ (say fat en) - the set of natural numbers including zero. 0, 1, 2 are members of this set.
  - $\varnothing$ - the empty set, the set with no members. Think of an empty bag - there is nothing in it.

  Z and $\mathbb{Z}$ are not the same thing. Z is the name of a notation that uses maths to specify computer systems. Z is the set of all whole numbers. Whole numbers are also known as *integ*

Number Range

- If a sequence of integers form a set, we can use number range notation.

- 1..7 (say 1 up to 7) is the set of all integers between 1 and 7 inclusive.
- 1..7 is the set { 1, 2, 3, 4, 5, 6, 7 }. Note that there are just two dots between the two integers that mark the beginning and end of the sequence.

Membership

- If we look at the set { 1988, 1992, 1996, 2000, 2004 } we can see that 2000 is an element of the set but 2001 is not. We write
- 2000 e { 1988, 1992, 1996, 2000, 2004 } where e means is-a-member-of

And

- 2001 ‰ { 1988, 1992, 1996, 2000, 2004 } where ‰ means is-not-a-member-of

Equality

- Two sets are equal if they both have exactly the same elements. For example { 1988, 1992, 1996, 2000, 2004 } = { 2000, 1996, 1988, 2004, 1992 }
- = means is-the-same-as.
- But if two sets do not have exactly the same elements, they are not equal. For example
  - { 1988, 1992, 1996, 2000, 2004} ≠ { 1988, 1992, 1996, 2000 } are not equal because 2004 is a member of one set but not the other.
  - ≠ means is-not-the-same-as.

Department of
Software Engineering

BASIC TYPES

Basic Types

- a person has a name, a date of birth and an address. If we do not need to be concerned with details such as title, forename, middle names and surname, we introduce the type *NAME*, the set of all possible names.
- If we do not need to bother with days, months, years and calendars (e.g. Chinese, Bengali and Gregorian) we introduce the type *DATE*, the set of all possible dates.
- If we do not need to bother with house number, street, city and postcode, we introduce the type *ADDRESS*, the set of all postal addresses.

Basic Types

- [ADDRESS, DATE, NAME]
  - We introduce as given sets *ADDRESS*, *DATE* and *NAME*. ADDRESS is the set of all postal addresses anywhere. DATE, the set of all possible dates in all possible calendars. NAME is the set of all names in full that any person might have.

Declarations

- PERSON, the set of all possible people on this planet, is pretty large. If we want to refer to just one of them we write

  aPerson : PERSON

  - aPerson represents just one, any one, of all the possible elements in PERSON.
- aPerson : PERSON is an example of a declaration.

Declarations - Parts

- A declaration has two parts.

  aPerson : PERSON

  - To the right of the colon is the name of a set; this name is PERSON.
  - To the left of the colon is a name for any element from that set; this name is aPerson. Since we do not necessarily know which element aPerson represents, aPerson is called a variable.
  - A variable has a name (e.g., aPerson) a type (e.g., PERSON) and a value taken from the type. Look at this declaration.
    - **aPersonsAge : 0..130**

Declarations – Parts (Example)

aPersonsAge : 0..130

- The name of the variable is aPersonsAge. Its type is $\mathbb{Z}$ because the values 0, 1, 2, and so on up to 130 are all elements from the larger set $\mathbb{Z}$. We think of a type as an inclusive set. Each element in the same set has the same type.

Naming Variables

- We choose our own names for variables. We choose descriptive names whenever clarity is required. For example.

  aPersonsAge : 0..130 rather than *a : 0..130*

- We choose a single letter when there is no doubt about clarity (but readers often appreciate more descriptive names).
  p : PERSON is ok but *aPerson : PERSON* might be preferable.

Naming Variables – Naming Convention

- We follow the convention that names we choose for our variables
  - start with a lower case letter - *person* not *Person*
  - are written entirely in lower case except the first letter of each word in the name – *aPersonsAge* not *apersonsage*
  - do not contain spaces, underscores or hyphens - *anAddress* not *an Address*
- Writing variable names in a mixture of lower and upper case (capital) letters helps us to tell them apart from type names, which are written entirely in upper case.

Consistency

- Every Z object is a type of one kind or another. This is important because it helps us discover inconsistencies in what we write.
  - For example, given the type

    [PERSON]

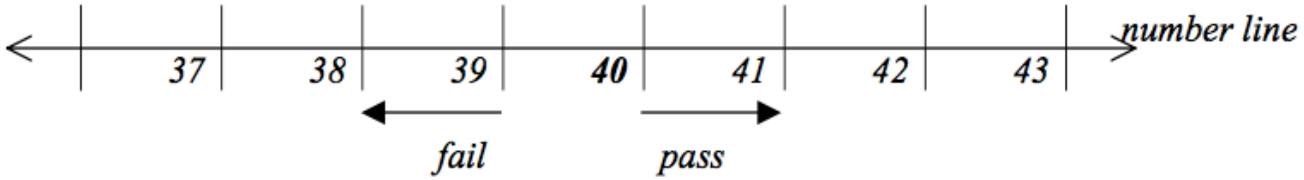
    and the declarations

    p, q : PERSON

    r: $\mathbb{Z}$
- p = q is consistent because both p and q are variables of type PERSON.
- But to say p = r is a nonsense because p is of type PERSON and r is of type integer. How can you say that a person and an integer are the same object?

The Numeric Comparison and Operators

- 40% is a critical mark in some exams. A mark of 40 or more is a pass. A mark of 39 or less is a fail. The boundary between fail and pass is shown on the number line below.



- 39 is-less-than 40. We write 39 < 40. < means is-less-than.
- 41 is-more-than 40. We write 41 > 40. > means is-more-than.
- < and > are known as comparison or relational operators.
- ≤ means less-than-or-equal-to. ≥ means more-than-or-equal-to.

Predicates

- The types of predicate we shall discuss include

  $=, \in$ equals and membership
  $<, >$ relations: less than and more than
  $\wedge$, $\vee$ connectives: conjunction (and) and disjunction (or)

- Putting the declaration and predicate together, as shown below, specifies a set.

  $$\{ n:\mathbb{Z} \mid n>0 \}$$

  - The declaration and predicate parts are separated by a | symbol.

- The declaration to the left of the | is the source of the elements of the set. The predicate to the right of the |.

The Connectives

- The connectives include conjunction (and) and disjunction (or). They allow us to connect smaller predicates together to form larger ones.
- For example, given the declaration $n : \mathbb{Z}$, here are two small predicates that say n is more than zero and n is less than 6.

  $n > 0$

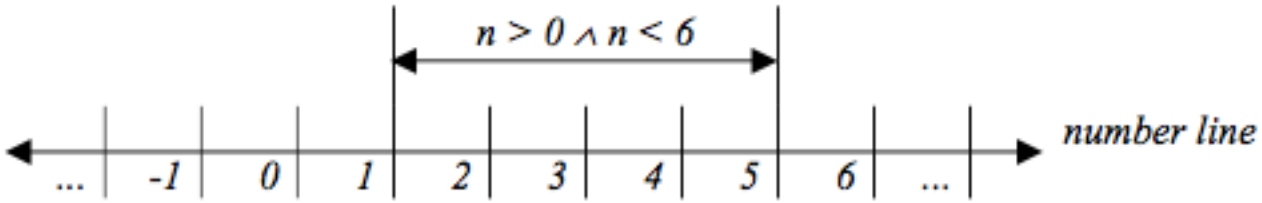
  $n < 6$
- Now join them together with a $\wedge$, which means and-at-the-same-time.

  $n > 0 \wedge n < 6$

The Connectives – conjunction

- We then have a composite predicate known as a conjunction. This predicate is true if n is more than 0 and, at the same time, n is less than 6.

$$\{ n : \mathbb{Z} \mid n > 0 \land n < 6 \} = \{ 1, 2, 3, 4, 5 \}$$

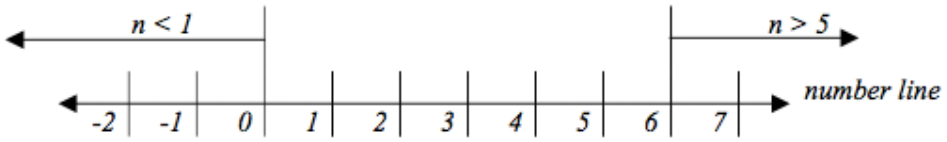


The Connectives – disjunction

- Here are two small predicates that say n **is** less than 1 and n is **more** than 5.

  n<1

  n>5



- Now join them together with ∨ , meaning **or**.

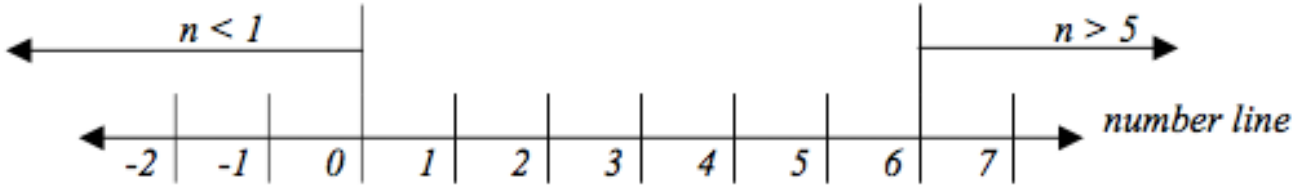
  n<1 ∨ n>5

The Connectives – disjunction

- We then have a composite predicate known as a disjunction. The predicate **is** true if either n **is** less than 1 or n **is** more than 5.

$\{ n : \mathbb{Z} \mid n < 1 \vee n > 5 \} = \{ ..., -2, -1, 0, 6, 7, 8, ... \}$

This is the set of all integers excluding { 1, 2, 3, 4, 5 }



It is easy to remember what the two symbols $\wedge$ and $\vee$ mean when you notice that the $\wedge$ looks a bit like the **A** in And.

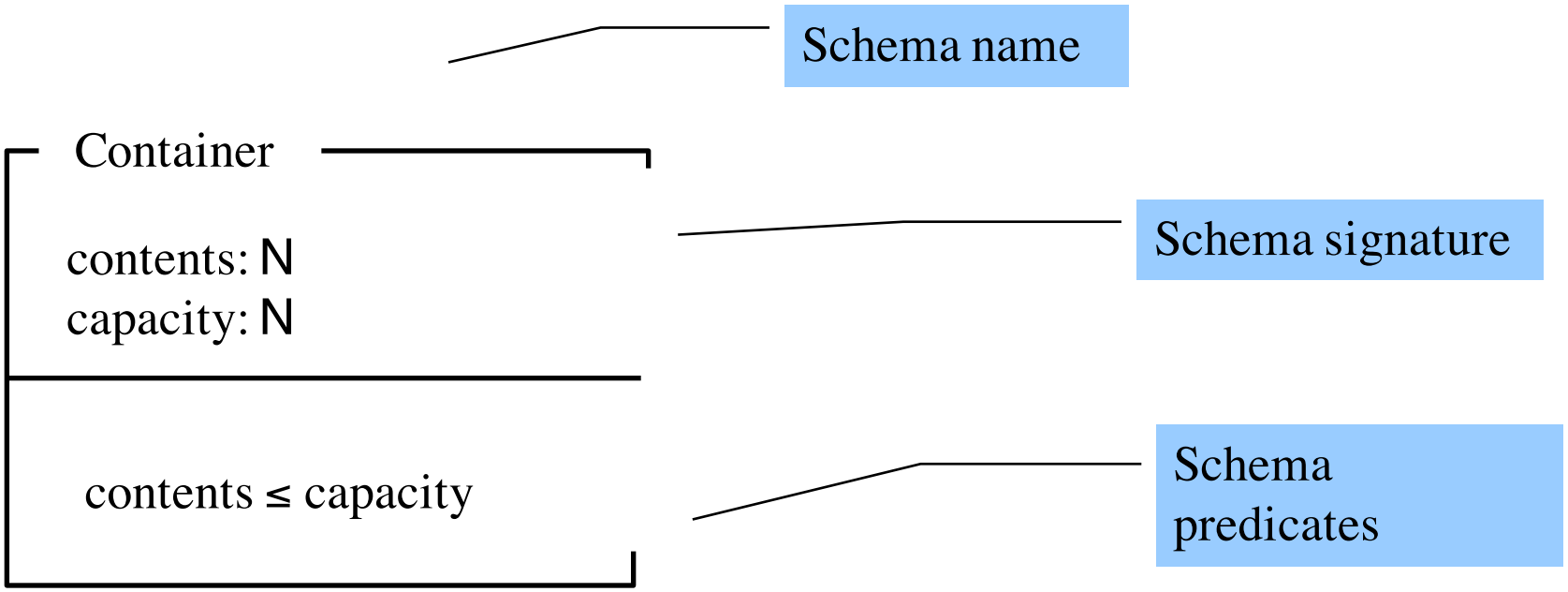
Department of
Software Engineering

SCHEMAS

Schemas

- Now we see how to combine **declarations** and **predicates** into **structures** called **schemas**.
- A schema represents a system's state.
- A collection of schemas models the behaviour of a computer system.
- **System**: The system we shall look at is a **simple counter**. A counter would be used, for example, to count the number of vehicles passing a census point, **the number of people entering a stadium** or the number of fleas in a bird's nest.
- **State**: The contents of a system's memory are called its state.

Z Schema



Schema predicates are always true
Predicates can refer only to elements in the signature

Schemas – System and State

- **System**: The system we shall look at is a **simple counter**. A counter would be used, for example, to count the number of vehicles passing a census point, **the number of people entering a stadium** or the number of fleas in a bird's nest.
- **State**: The contents of a system's memory are called its state. For our counter system that would be the current value of the count together with the maximum it can reach.
  - We would have the initial state when the count is zero, maximum is 9999 (say).
  - Initial state: count = 0, maximum = 9999

Schemas – State (Interim and End State)

- We might have an **interim state** when the count is between zero and the maximum value that count can have.

  Interim state: count = 147, maximum = 9999

- We might have an **end state** when the count has reached it maximum value and cannot be advanced any further.

  End state: count = 9999, maximum = 9999

- Effectively, the state of a system is the collection of values stored in its variables.

State Space Schemas

- Here is an example state-space schema, representing part of a system that records details about the phone numbers of staff. (Assume that *NAME* is a set of names, and *PHONE* is a set of phone numbers.)

$$
\begin{array}{|l}
\textit{PhoneBook} \\
\hline
known : \mathbb{P}\, NAME \\
tel : NAME \pfun PHONE \\
\hline
\mathrm{dom}\, tel = known \\
\hline
\end{array}
$$

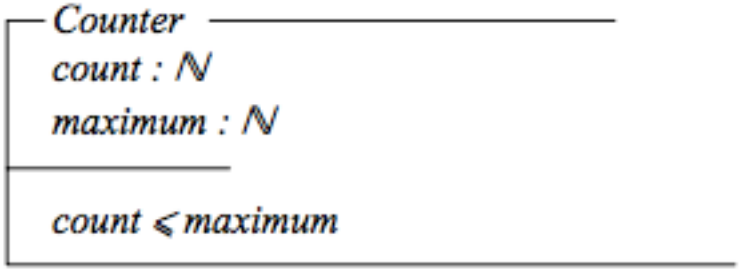
State Space Schemas

- The declarations part of this schema introduces two variables: ***known*** and ***tel***.
- The value of ***known*** will be a subset of *NAME*, i.e., a set of names. [This variable will be used to represent all the names that we know about — those that we can give a phone number for.]
- The value of *tel* will be a partial function from *NAME* to *PHONE*, i.e., it will associate names with phone numbers.
- The domain of *tel* is always equal to the set *known*.

$$
\begin{array}{|l}
\textit{PhoneBook} \\
\hline
known : \mathbb{P}\, NAME \\
tel : NAME \pfun PHONE \\
\hline
\mathrm{dom}\, tel = known \\
\hline
\end{array}
$$

System State Schemas

- A counter has just two variables, one that we shall name count, one that we shall name maximum. count can equal, but never exceed, maximum. Variables are derived from what an object has.
- A schema is a set of variables together with a set of predicates constraining those variables. A schema is drawn as an open box - see below.



System State Schemas

Our schema, named Counter and shown above, has two variables, count and maximum, and one predicate $count \leq maximum$.

- We (nearly) always start schema names with an uppercase letter. A schema name cannot contain spaces.

Schemas – The Initial State

We describe the state the system is in when it is first started. The *InitCounter* schema defined below describes the initial state of Counter.

InitCounter

Counter

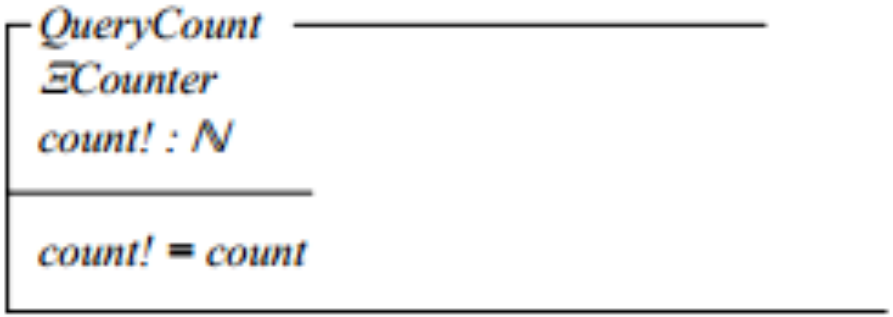
$count = 0$
$maximum = 9999$

InitCounter

$count : \mathbb{N}$
$maximum : \mathbb{N}$

$count \leqslant maximum$
$count = 0$
$maximum = 9999$

Query System State



Ξcounter (say Xi Counter) says include all the variables and predicates defined in the Counter schema; the values stored in these variables will not change.

The declaration count! : N says count output is drawn from the set N. The **!** mark stands for

output.

The predicate count! = count says count output is the same as the (system variable) count.

Change System State

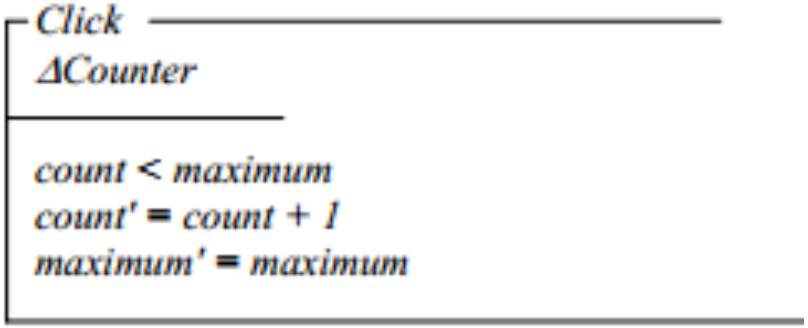
- Not only does a schema define and report on a system's state, it also describes changes in that state.
- The value of count does not remain the same forever. From time to time the counter will be clicked and the value stored in count will be moved on. The Click schema shown below updates the count system state variable but leaves the maximum variable unchanged.

Click

$\Delta Counter$

$count < maximum$
$count' = count + 1$
$maximum' = maximum$

Change System State



- *ΔCounter* (say delta Counter) says
  - include all the variables and predicates defined in the *Counter* schema
  - the values stored in some (or all) of these variables may well change
  - decorate each variable with a ' to represent the state after an update has taken place

Example 1

A student may join the class if the class is not already full and if the student has not already enrolled. A new student cannot have passed all their assignments.

Example

A student may join the class if the class is not already full and if the student has not already enrolled. A new student cannot have passed all their assignments.

EnrolOk

$\Delta Class$

$student? : STUDENT$

$\#enrolled < maxClassSize$

$student? \notin enrolled$

$enrolled' = enrolled \cup \{ student? \}$

$passed' = passed$

The *?* in *student?* means input.

Example 2

An existing student is transferred to the passed set provided they have passed all their assignments and have not already been transferred. Every student in passed must also be in enrolled.

Example 2

An existing student is transferred to the passed set provided they have passed all their assignments and have not already been transferred. Every student in passed must also be in enrolled.

CompleteOk

$\Delta Class$

$student? : STUDENT$

$student? \in enrolled$

$student? \notin passed$

$enrolled' = enrolled$

$passed' = passed \cup \{ student? \}$

Example 2

Only those existing students who have passed may leave with a certificate

LeaveWithCertificateOk

$\Delta Class$

$student? : STUDENT$

$student? \in passed$

$enrolled' = enrolled \setminus \{ student? \}$

$passed' = passed \setminus \{ student? \}$

Success and Error Schemas

The Success schema has just one declaration and one predicate. It will be combined with other schemas to indicate their successful outcome.

Success

$report! : REPORT$

$report! = success$

The class is full if the number of enrolled students has reached (or by some mistake has exceeded) the maximum class size.

ClassFull

$\Xi Class$

$report! : REPORT$

$\#enrolled \geq maxClassSize$

$report! = classFull$

Example 3

A student cannot be enrolled again if they are already enrolled.

AlreadyEnrolled

$\Xi Class$

$student? : STUDENT$

$report! : REPORT$

$student? \in enrolled$

$report! = alreadyEnrolled$

Example 4

If a student is not enrolled they cannot be passed.

NotEnrolled

$\Xi Class$

$student? : STUDENT$

$report! : REPORT$

$student? \notin enrolled$

$report! = notEnrolled$

Example 5

The same student cannot be passed twice.

AlreadyPassed

$\Xi Class$

$student? : STUDENT$

$report! : REPORT$

$student? \in passed$

$report! = alreadyPassed$

A student who has not passed cannot leave with a certificate.

NotPassed

$\Xi Class$

$student? : STUDENT$

$report! : REPORT$

$student? \notin passed$

$report! = notPassed$

- A hotel maintains a record of the current state of its rooms, whether occupied or not. Write and explain a Z specification for the system described below.

Use Case:	Commission
Purpose:	to add a new room to the hotels rooms for guests system
Pre-conditions:	the room has not already been added
Initiating Actor:	accommodation manager
1	manager inputs details of the new room
2	system confirms new room added
3	exit success
Exceptions	
2a	room already in the system
2a1	exit failure

REFERENCES

References/1

•

Department of
Software Engineering

QUESTIONS ?

Daffodil International University

Department of Software Engineering

thank you!