FIT3037
Software Engineering

Specification Using Z

Z Schemas

The basic representation of schemas in Z:

```
── Name ──────────────────────────────
|
| Declarations
|
|─────────────────────
|
| Predicates
|
|──────────────────────────────────────
```

Variable declarations of the form
variable : type

Give the properties of, and relationships between variables

Types in Z

Data types

- Integers $\mathbb{Z}$ (-ve, 0 and +ve whole numbers - http://en.wikipedia.org/wiki/Integer)
- Natural Numbers $\mathbb{N}$ (0 and +ve whole numbers - http://en.wikipedia.org/wiki/Natural_numbers)
- Positive Integers $\mathbb{N}_1$ (+ve whole numbers)
  - Actually both are subsets of the basic integer $\mathbb{Z}$ type
  - So x: $\mathbb{N}$ is equivalent to {x:Z| x $\geq$ 0} and x: $\mathbb{N}_1$ is equivalent to {x:Z| x > 0}
- Can also declare ranges

  day:1 .. 7 is equivalent to { day: $\mathbb{Z}$ | day $\geq$1 $\wedge$ day $\leq$ 7 }

- Note that Z does not include basic types such as Reals, Booleans and Characters

Types in Z (2)

- User Defined Types
  - User can define any type they like
    - [NAME, PHONE_NUMBER]
      - So NAME is an abstract type which can hold a single one of all the possible names
      - Must be defined before they are used
      - By convention written in all capitals
      - No need to actually declare internal elements or size
- Free Types
  - FreeType ::= Element_1 | Element_2 | … | Element_n
  - Used for defining enumerated types etc
  - Each value must be a distinct literal (disjoint value)
    - RESULT ::= OK | WARNING | ERROR
    - DIGIT ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
      - Equivalent to DIGIT == {‘0’ , ‘1’ , ‘2’ , ‘3’ , ‘4’ , ‘5’ , ‘6’ , ‘7’ , ‘8’ , ‘9’ }

Types in Z (3)

- Cartesian Products
  - NAME ⊠ ADDRESS ⊠ TEL_NO
    - Is the set of all three-tuple subsets of NAME, ADDRESS, TEL_NO

- Tuples
    - Tuple shown as elements separated by commas within round brackets (x_1, x_2, x_3,…, x_n)
    - To select one element can use the tuple selection operator
      tuple.index

  - Example
    - t == (Ray Smith, Salisbury Lane, 26462)
    - t.1 = Ray Smith
    - t.2 = Salisbury Lane
    - t.3 = 26462

Global Variables

- We can define global variables by declaring *axiomatic definitions*

  ✹declaration
  ⇮♉♉♉♉♉♉
  ✹predicate

  - Example

    ✹range : ☠
    ⇮♉♉♉♉♉♉♉♉♉♉♉♉
    ✹0 ⬂ range ❍ range ⬂ 10

  - Note that the predicate part is optional

    ✹global_constant : TYPE

eg

✹range : ☠

Birthday Book Example

This example is taken from Chapter 1 of
http://spivey.oriel.ox.ac.uk/~mike/zrm/

It is a classic simple example that allows us to demonstrate the basic capabilities of the Z specification language without becoming overly detailed and difficult to understand.

The author is Mike Spivey a very well respected researcher in the formal methods field and a member of the team that developed the Z language.

The State Space for a Birthday Book

The state space for the implementation:

```
[NAME, DATE]

── BirthdayBook ─────────────
│ known: ℙ NAME
│ birthday: NAME ⇸ DATE
│
├───────────────────────
│
│ known = dom birthday
│
└─────────────────────────────
```

A possible state of the system…

known = { John, Mike, Susan }

birthday = { John ↦ 25-Mar-70,
Mike ↦ 20-Dec-60,
Susan ↦ 15-Apr-65 }

domain **range**

NB: Important to understand why we need the powerset in the declaration of known. The type of Known is ℙ NAME as it contains a set from the powerset of NAME at any one time. birthday is a function which returns the birthday of a given name.

Also why the predicate is needed to maintain consistency between the two variables

AddBirthday Function

```
AddBirthday
ΔBirthdayBook
name? : NAME
date? : DATE

name? ∉ known
birthday′ = birthday ∪ { name? ➊ date? }
```

Δ = state of the system changes (greek Delta)

x? is an input to the system

So basically we are saying:

- ***we want to use the BirthdayBook statespace already declared***
- ***the contents of BirthdayBook are allowed to be changed in this schema***
- ***we need two inputs from the user; the name and birthdate of the person***
- ***we check that the name is not already entered in the set of known names***
- ***then we use the union operator to add to the birthday function the new name and date pair***

FindBirthday Function

```
── FindBirthday ──────────────────
ΞBirthdayBook
name? : NAME
date! : DATE
──────────────────
name? ∈ known
date! = birthday(name?)
──────────────────────────────────
```

Ξ = state of the system does **NOT** change (Ξ = greek Xi)

x! = is an output from the system

Remind Function

```
 ── Remind ─────────────────────────────
| ΞBirthdayBook
| today? : DATE
| cards! : ℙ NAME
|──────────────────────
| cards! = {n:known | birthday(n)= today?}
|
 ───────────────────────────────────────
```

y is a member of the set { x: S |… x … }
 if (and only if) y is a member of S
 and the … y … is satisfied
(condition is obtained by replacing x with y)

Initialising the System

```
InitBirthdayBook
ΔBirthdayBook
─────────────
known = ∅
```

ie known is empty

… therefore the function birthday is also empty due to the original predicate in the state space schema

Handling Errors (1)

- Previous specifications assume no errors
  - ie pre-conditions are always true
- What if errors occur?
  - Options:
    - Scrap the specification and start again
    - Add to the existing specification
- Using the second approach:
  - Describe separately errors and responses to them
  - Use the Z schema calculus to combine specifications

Handling Errors (2)

- Add an extra output result of type REPORT where

```
REPORT ::= ok|already_known|not_known
```

    So we are declaring a type called REPORT which may only
    have one of these three defined values

- For producing output ok, already_known, and not_known, we need to define the following three respective schemas:
  - Success
  - AlreadyKnown
  - NotKnown

Success Schema

```
 ── Success ──────────────────────
 result! : REPORT
 ─────────────────────
 result! = ok
 ────────────────────────────────
```

ie whenever the operation is successful (it always is in this case) we just need to output OK

AlreadyKnown Schema

```
── AlreadyKnown ──────────────────────
ΞBirthdayBook
name? : NAME
result! : REPORT
──────────────────

name? ∈ known
result! = already_known
──────────────────────────────────────
```

Combining with AddBirthday …

$\Downarrow$ Introduces a new schema ie "is defined as"

RAddBirthday $\Downarrow$ (AddBirthday ∧ Success)
∨ AlreadyKnown

NotKnown Schema

```
┌── NotKnown ──────────────────────────
│ ΞBirthdayBook
│ name? : NAME
│ result! : REPORT
├──────────────────────
│ name? ∉ known
│ result! = not_known
└──────────────────────────────────────
```

Robust version of FindBirthday …

```
RFindBirthday ≙ ( FindBirthday ∧ Success )
                ∨ NotKnown
```

Examples

- ***Develop a formal expression using logic and set operators for the statement:***

"Every train driver has one railway crossing which they fear the most"

Examples

•***Develop a formal expression using logic and set operators for the statement:***

"Every train driver has one railway crossing which they fear the most"

Use the predicates TRAINDRIVER (x), CROSSING (x) and MOSTFEARED (x,y) to represent the "is train driver" , "is a crossing" and "y is most feared crossing of person x" functions.

Examples

"Every train driver has one railway crossing which they fear the most"

Use the predicates TRAINDRIVER (x), CROSSING (x) and MOSTFEARED (x,y) to represent the "is train driver" , "is a crossing" and "y is most feared crossing of x" functions.

$$\forall x \ \exists y \ (TRAINDRIVER(x) \Rightarrow CROSSING(y) \wedge MOSTFEARED(x,y))$$

Example 2

You are to produce a formal specification for a weather map system that records the average yearly rainfall for a series of named regions. The following Z schema defines the state space for this system.

[Region, Real]

——— ***WeatherMap*** ———

known: $\mathbb{P}$ ***Region***

rainfall: ***Region*** $\rightharpoonup$ ***Real***

———————————

dom rainfall = known

———————————

(where Real is the type that can contain one of the set of all floating point numbers)

Example 2

Write a schema for an operation AddRegion that takes a region not currently recorded in the weather map and adds its average rainfall.

Example 2

Write a schema for an operation AddRegion that takes a region not currently recorded in the weather map and adds its average rainfall.

Ans:

AddRegion

ΔWeatherMap

region? : Region

avg? : Real

region? ∉ known

rainfall' = rainfall ∪ { region? ❶ avg? }

What other predicate might we want to put in this schema before saving the new pair?

Example 3

Write a schema for an operation Total2Regions that takes two regions from the user and outputs the combined average rainfall for these regions.

Example 3

Write a schema for an operation Total2Regions that takes two regions from the user and outputs the combined average rainfall for these regions.

Ans:

Total2Regions

⌧WeatherMap

reg1? : Region

reg2? : Region

total! : Real

reg1? ♏ known ❍ reg2 ♏ known

total! = rainfall(reg1) + rainfall(reg2)

Example 3 cont

How could we use the schema calculus to make Total2Regions robust? That is handle any errors that may occur.

Ans:

Error::= Region 1 Not Known | Region 2 Not Known

UnknownRegion1

Ξ***WeatherMap***

reg1? : Region

error! : ERROR

---

reg1? $\notin$ ***known***

error! = Region 1 Not Known

UnknownRegion2

Ξ***WeatherMap***

reg2? : Region

error! : ERROR

---

reg2? $\notin$ ***known***

error! = Region 2 Not Known

RTotal2Regions $\widehat{=}$ ***Total2Regions*** $\lor$ ***(UnknownRegion1*** $\lor$ ***UnknownRegion2)***

Useful reading

- *Up to Section 1.3 of Chapter 1 from* ***spivey.oriel.ox.ac.uk/mike/zrm/zrm.pdf***
- *Section 4.2 Z Specification language of Study Guide 3*