Model-based Specification

- Formal specification of software by developing a mathematical model of the system

Objectives

- To introduce an approach to formal specification based on mathematical system models
- To present some features of the Z specification language
- To illustrate the usefulness of Z by describing small examples
- To show how Z schemas may be used to develop incremental specifications
Topics covered

- Z schemas
- The Z specification process
- Specifying ordered collections

Model-based specification

- Defines a model of a system using well-understood mathematical entities such as sets and functions.
- The state of the system is not hidden (unlike algebraic specification).
- State changes are straightforward to define.
- VDM and Z are the most widely used model-based specification languages.
Z as a specification language

- Based on typed set theory
- Probably now the most widely-used specification language
- Includes schemas, an effective low-level structuring facility
- Schemas are specification building blocks
- Graphical presentation of schemas make Z specifications easier to understand

Z schemas

- Introduce specification entities and defines invariant predicates over these entities
- A schema includes
  - A name identifying the schema
  - A signature introducing entities and their types
  - A predicate part defining invariants over these entities
- Schemas can be included in other schemas and may act as type definitions
- Names are local to schemas
Z schema highlighting

<table>
<thead>
<tr>
<th>Schema name</th>
<th>Schema signature</th>
<th>Schema predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>contents: N</td>
<td>capacity: N</td>
</tr>
<tr>
<td></td>
<td>contents ≤ capacity</td>
<td></td>
</tr>
</tbody>
</table>

An indicator specification

<table>
<thead>
<tr>
<th>Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>light: {off, on}</td>
</tr>
<tr>
<td>reading: N</td>
</tr>
<tr>
<td>danger_level: N</td>
</tr>
</tbody>
</table>

light = on ⇔ reading ≤ danger_level
Storage tank specification

```
Storage_tank
  Container
  Indicator
    reading = contents
    capacity = 5000
    danger_level = 50
```

Full specification of a storage tank

```
Storage_tank
  contents: \leq
  capacity: \leq
  reading: \leq
  danger_level: \leq
  light: \{off, on\}

contents \leq capacity
light = on \iff reading \leq danger_level
reading = contents
capacity = 5000
danger_level = 50
```
Z conventions

- A variable name decorated with a quote mark (N') represents the value of the state variable N after an operation.
- A schema name decorated with a quote mark introduces the dashed values of all names defined in the schema.
- A variable name decorated with a ! represents an output.

Z conventions

- A variable name decorated with a ? represents an input.
- A schema name prefixed by the Greek letter Xi (Ξ) means that the defined operation does not change the values of state variables.
- A schema name prefixed by the Greek letter Delta (Δ) means that the operation changes some or all of the state variables introduced in that schema.
Operation specification

- Operations may be specified incrementally as separate schema then the schema combined to produce the complete specification
- Define the ‘normal’ operation as a schema
- Define schemas for exceptional situations
- Combine all schemas using the disjunction (or) operator

A partial spec. of a fill operation

<table>
<thead>
<tr>
<th>Fill-OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \text{ Storage_tank} )</td>
</tr>
<tr>
<td>amount?: ( \mathbb{N} )</td>
</tr>
<tr>
<td>contents + amount? ( \leq ) capacity</td>
</tr>
<tr>
<td>contents’ = contents + amount?</td>
</tr>
</tbody>
</table>
Storage tank fill operation

OverFill

Ξ Storage-tank amount? : \( r \)
\( r! : \text{seq CHAR} \)

capacity < contents + amount?
\( r! = "\text{Insufficient tank capacity – Fill cancelled}" \)

Fill

Fill-OK \( \lor \) OverFill

The Z specification process

Write informal specification \( \rightarrow \) Decompose system \( \rightarrow \) Specify system components \( \rightarrow \) Compose component specifications

Define given sets and types \( \rightarrow \) Define state variables \( \rightarrow \) Define initial state \( \rightarrow \) Define 'correct' operations \( \rightarrow \) Define exceptional operations \( \rightarrow \) Combine operation schemas
Data dictionary specification

- Data dictionary, introduced in Chapter 6, will be used as an example. This is part of a CASE system and is used to keep track of system names.

- Data dictionary structure
  - Item name
  - Description
  - Type. Assume in these examples that the allowed types are those used in semantic data models
  - Creation date

Given sets

- Z does not require everything to be defined at specification time
- Some entities may be ‘given’ and defined later
- The first stage in the specification process is to introduce these given sets
  - [NAME, DATE]
  - We don’t care about these representations at this stage
Type definitions

- There are a number of built-in types (such as INTEGER) in Z
- Other types may be defined by enumeration
  - `Sem_model_types = { relation, entity, attribute }`
- Schemas may also be used for type definition. The predicates serve as constraints on the type

Specification using functions

- A function is a mapping from an input value to an output value
  - `SmallSquare = {1 → 1, 2 → 4, 3 → 9, 4 → 16, 5 → 25, 6 → 36, 7 → 49 }`
- The domain of a function is the set of inputs over which the function has a defined result
  - `dom SmallSquare = {1, 2, 3, 4, 5, 6, 7 }`
- The range of a function is the set of results which the function can produce
  - `rng SmallSquare = {1, 4, 9, 16, 25, 36, 49 }`
The function SmallSquare

<table>
<thead>
<tr>
<th>Domain (SmallSquare)</th>
<th>Range (SmallSquare)</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>4</td>
</tr>
<tr>
<td>three</td>
<td>9</td>
</tr>
<tr>
<td>four</td>
<td>16</td>
</tr>
<tr>
<td>five</td>
<td>25</td>
</tr>
<tr>
<td>six</td>
<td>36</td>
</tr>
<tr>
<td>seven</td>
<td>49</td>
</tr>
</tbody>
</table>

Data dictionary modeling

- A data dictionary may be thought of as a mapping from a name (the key) to a value (the description in the dictionary)

- Operations are
  - Add. Makes a new entry in the dictionary or replaces an existing entry
  - Lookup. Given a name, returns the description.
  - Delete. Deletes an entry from the dictionary
  - Replace. Replaces the information associated with an entry
Data dictionary entry

```
DataDictionaryEntry

  entry: NAME
  desc: seq char
  type: Sem_model_types
  creation_date: DATE

#description ≤ 2000
```

Data dictionary as a function

```
DataDictionary

  DataDictionaryEntry
  ddict: NAME → {DataDictionaryEntry}
```
Data dictionary - initial state

<table>
<thead>
<tr>
<th>Init-DataDictionary</th>
<th>DataDictionary'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ddct' = Ø</td>
</tr>
</tbody>
</table>

Add and lookup operations

Add_OK

Δ DataDictionary
name?: NAME
entry?: DataDictionaryEntry

name? ∈ dom ddct
ddct' = ddct ∪ {name? → entry?}

Lookup_OK

⊆ DataDictionary
name?: NAME
entry!: DataDictionaryEntry

name? ∈ dom ddct
entry! = ddct (name?)
Add and lookup operations

Add_Error

\[ \exists \text{DataDictionary} \]
name? : NAME
error!: seq char

name? \in dom ddict
error! = "Name already in dictionary"

Lookup_Error

\[ \exists \text{DataDictionary} \]
name? : NAME
error!: seq char

name? \notin dom ddict
error! = "Name not in dictionary"

Function over-riding operator

ReplaceEntry uses the function overriding operator (written \( \oplus \)). This adds a new entry or replaces an existing entry.

- phone = \{ Ian \rightarrow 3390, Ray \rightarrow 3392, Steve \rightarrow 3427\}
- The domain of phone is \{Ian, Ray, Steve\} and the range is \{3390, 3392, 3427\}.
- newphone = \{Steve \rightarrow 3386, Ron \rightarrow 3427\}
- phone \oplus newphone = \{ Ian \rightarrow 3390, Ray \rightarrow 3392, Steve \rightarrow 3386, Ron \rightarrow 3427\}
Replace operation

Replace_OK

Δ DataDictionary
name?: NAME
entry?: DataDictionaryEntry

name? ∈ dom ddict
ddict' ≈ {name? → entry?}

Deleting an entry

Uses the domain subtraction operator (written \(\setminus\)) which, given a name, removes that name from the domain of the function

- phone = \{ Ian \rightarrow 3390, Ray \rightarrow 3392, Steve \rightarrow 3427\}
- \{Ian\} \setminus phone
- \{Ray \rightarrow 3392, Steve \rightarrow 3427\}
Delete entry

Δ DataDictionary
name?: NAME

name? ∈ dom ddict
ddict' = {name?} ⊄ ddict

Specifying ordered collections

⊗ Specification using functions does not allow ordering to be specified
⊗ Sequences are used for specifying ordered collections
⊗ A sequence is a mapping from consecutive integers to associated values
A Z sequence

<table>
<thead>
<tr>
<th>Domain (SqSeq)</th>
<th>Range (SqSeq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
</tr>
</tbody>
</table>

Data dictionary extract operation

- The Extract operation extracts from the data dictionary all those entries whose type is the same as the type input to the operation.
- The extracted list is presented in alphabetical order.
- A sequence is used to specify the ordered output of Extract.
The Extract operation

Extract

DataDictionary
rep!: seq {DataDictionaryEntry}
in_type?: Sem_model_types

∀n : dom ddict • ddict(n). type = in_type? ⇒ ddict(n) ∈ rng rep!
∀i : 1 ≤ i ≤ #rep! • rep!(i).type = in_type?
∀i : 1 ≤ i ≤ #rep! • rep!(i) ∈ rng ddict
∀i, j: dom rep! • (i < j) ⇒ rep!.name(i) < NAME rep!.name (j)

Extract predicate

- For all entries in the data dictionary whose type is in_type?, there is an entry in the output sequence
- The type of all members of the output sequence is in_type?
- All members of the output sequence are members of the range of ddict
- The output sequence is ordered
Data dictionary specification

The Data Dictionary

- DataDictionary
- Init-DataDictionary
- Add
- Lookup
- Delete
- Replace
- Extract

Key points

- Model-based specification relies on building a system model using well-understood mathematical entities
- Z specifications are made up of mathematical model of the system state and a definition of operations on that state
- A Z specification is presented as a number of schemas
- Schemas may be combined to make new schemas
Key points

- Operations are specified by defining their effect on the system state. Operations may be specified incrementally then different schemas combined to complete the specification.

- Z functions are a set of pairs where the domain of the function is the set of valid inputs. The range is the set of associated outputs. A sequence is a special type of function whose domain is the consecutive integers.