Formal Specification

Approaches to Formal Specification

• Functional
  – The system is described as a number of functions.
  – Unnatural and complex for large systems

• Algebraic
  – The system is described in terms of operations and their relationships.

• Model-based
  – A model of the system is constructed using well-understood mathematical entities such as sets and sequences
Algebraic Specification

- Based on concept of abstract data types
- Large systems are usually decomposed into sub-systems which are accessed through a defined interface.

Sub-system A ➔ Sub-system B

Algebraic Specification (2)

Components of a specification:
- Specification name and generic parameter list
- Type name and declaration of imported specifications
- Informal description
- Operation signatures
- Axioms that define the operational semantics
Array Specification

ARRAY (Elem: [Undefined \rightarrow Elem])

sort Array
imports Integer

Arrays are collections of elements of generic type Elem. They have a lower and upper bound...

Create(Integer, Integer) \rightarrow Array
Assign(Array, Integer, Elem) \rightarrow Array
First(Array) \rightarrow Integer
Last(Array) \rightarrow Integer
Eval(Array, Integer) \rightarrow Elem

Array Specification Axioms

First(Create(x, y)) = x
First(Assign(a, n, v)) = First(a)
Last(Create(x, y)) = y
Last(Assign(a, n, v)) = Last(a)
Eval(Create(x, y), n) = Undefined
Eval(Assign(a, n, v), m) =
    if m < First(a) or m > Last(a) then Undefined
    else if m = n then v
    else Eval(a,m)

Eval(
    Assign(
        Assign(
            Create(1,10), 1, 5), 2, 10), 1) = 5
Binary Tree Specification

Signatures:

Create → BT
Add(BT, Elem) → BT
Left(BT) → BT
Right(BT) → BT
Data(BT) → Elem
Empty(BT) → Boolean
Contains(BT, Elem) → Boolean
Build(BT, Elem, BT) → BT

Binary Tree Specification (2)

Axioms:

Add(Create, E) = Build(Create, E, Create)
Add(B, E) = if E < Data then Add(Left(B), E)
    else Add(Right(B), E)
Left(Create) = Create
Data(Create) = Undefined
Left(Build(L, D, R)) = L
Data(Build(L, D, R)) = D
Empty(Create) = true
Empty(Build(L, D, R)) = false
Contains(Create, E) = false
Contains(Build(L, D, R), E) =
    if E = D then true else
        if E < D then Contains(L, D) else
            Contains(R, D)
Systematic Specification

- Specification structuring
  - The informal interface specification is structured into a set of abstract data types with proposed operations.

- Specification naming
  - Name, generic parameters, type

- Operation selection
  - Constructors, Manipulators, Inspectors

- Informal operation specification

- Syntax definition

- Axiom definition

Structured Specification: Reuse

- Instantiation of generic specifications
  - Array<Integer>, Array<Array<Character>>, etc.

- Incremental development of specifications
  - Import specification of simple types and use them in the definition of more complex types, e.g. the specification of Coordinate might be used in the specification of Cursor.

- Enrichment of specifications
  - Inheritance of signatures and axioms from a base specification
Error Specification

Several approaches:

- A special constant operation, such as *Undefined*, may be defined.

- Operations may be defined to evaluate to *tuples*, where one element of the tuple is a boolean that indicates successful evaluation.

- The specification may include an exceptions section which defines conditions under which the axioms do not hold.

---

Model-based Specification with Z

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
• The illustrate the use of Z using small examples
• To show how Z schemas may be used to develop incremental specifications

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
Z 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
An *Indicator* Specification

```
Indicator
  light: \{ off, on \}
  reading: \mathcal{N}
  danger_level: \mathcal{N}

light = on \iff reading \leq danger_level
```
Storage Tank Specification

- Storage_tank
  - Container
  - Indicator

  reading = contents
  capacity = 5000
  danger_level = 50

Complete Storage Tank Specification

- Storage_tank
  - contents: \( \mathcal{N} \)
  - capacity: \( \mathcal{N} \)
  - light: \{ off, on \}
  - reading: \( \mathcal{N} \)
  - danger_level: \( \mathcal{N} \)

  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 dash (quote mark), $N'$, represents the value of the state variable $N$ after an operation.
- A schema name decorated with a dash 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 ($\Xi$) means that the defined operation does not change the values of state variables.
- A schema name prefixed by the Greek letter Delta ($\Delta$) 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>ΔStorage_tank</td>
</tr>
<tr>
<td>amount?: ( \mathcal{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

∃Storage_tank
amount?): \(\mathcal{N}\)
r!: seq CHAR

capacity < contents + amount?
r! = “Insufficient tank capacity – Fill canceled”

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

• A data dictionary 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 UML Class Diagrams
  – Creation date

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.
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
  – ModelElement = { class, field, method, association }
• Schemas may also be used for type definition. The predicates serve as constraints on the type
Data Dictionary Entry

- DataDictionaryEntry
  - name: NAME
  - description: seq char
  - type: ModelElement
  - creation_date: DATE

  #description ≤ 2000

---

Specification Using Functions

- A function is a mapping from an input value to an output value, e.g:
  - SmallSquare = \{1\rightarrow 1, 2\rightarrow 4, 3\rightarrow 9, 4\rightarrow 16, 5\rightarrow 25, 6\rightarrow 36, 7\rightarrow 49 \}
  - 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
  - Domain SmallSquare = \{1, 2, 3, 4, 5, 6, 7 \}

- The range of a function is the set of results which the function can produce
  - Range SmallSquare = \{1, 4, 9, 16, 25, 36, 49 \}
Set and Function Notation

• We can use set and function notations interchangeably, depending on what is more convenient.

• Function notation:
  \[ \text{SmallSquare}(3) = 9 \]

• Set notation:
  \[ (3, 9) \in \text{SmallSquare} \]
Data dictionary as a function

\[
\text{DataDictionary} \\
\text{DataDictionaryEntry} \\
ddict: \text{NAME} \mapsto \{ \text{DataDictionaryEntry} \}
\]

Data dictionary initialization

\[
\text{Init\_DataDictionary} \\
\text{DataDictionary'} \\
ddict' = \{ \}
\]
Add and lookup operations

---

**Add_OK**

\[ \Delta \text{DataDictionary} \]

\[ \text{entry?: DataDictionaryEntry} \]

\[ \text{entry?.name} \notin \text{dom ddict} \]

\[ \text{ddict'} = \text{ddict} \cup \{ \text{entry?.name } \rightarrow \text{entry?} \} \]

---

**Lookup_OK**

\[ \exists \text{DataDictionary} \]

\[ \text{name?: NAME} \]

\[ \text{entry!: DataDictionaryEntry} \]

\[ \text{name?} \in \text{dom ddict} \]

\[ \text{entry!} = \text{ddict(name?)} \]

---

Add and lookup operations

---

**Add_Error**

\[ \exists \text{DataDictionary} \]

\[ \text{entry?: DataDictionaryEntry} \]

\[ \text{error!: seq char} \]

\[ \text{entry?.name} \in \text{dom ddict} \]

\[ \text{error!} = \text{“Name already in dictionary”} \]

---

**Lookup_Error**

\[ \exists \text{DataDictionary} \]

\[ \text{name?: NAME} \]

\[ \text{error!: seq char} \]

\[ \text{name?} \notin \text{dom ddict} \]

\[ \text{error!} = \text{“Name not in dictionary”} \]
Function over-riding operator

- ReplaceEntry uses the function overriding operator (written $\oplus$). This adds a new entry or replaces and 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

\[
\Delta \text{DataDictionary} \\
\text{entry?: DataDictionaryEntry} \\
\text{entry?.name} \in \text{dom ddict} \\
ddict' = ddict \oplus \{ \text{entry?.name} \rightarrow \text{entry?} \} 
\]
Deleting an Entry

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

\[
\text{phone} = \{ \text{Ian} \rightarrow 3390, \text{Ray} \rightarrow 3392, \text{Steve} \rightarrow 3427 \} \\
\{\text{Ian}\} \triangle \text{phone} = \{\text{Ray} \rightarrow 3392, \text{Steve} \rightarrow 3427\}
\]

Delete Entry Operation

```
Delete_OK
Δ DataDictionary
name?: NAME

name? ∈ dom ddict
ddict’ = { name? } \triangle ddict
```
Specifying Ordered Collections

- Specification using sets does not allow ordering to be specified.
- Sequences are used for specifying ordered collections.
- A sequence is a function mapping consecutive integers to associated values.

A Z Sequence

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
</tbody>
</table>

\{ (1,a), (2,b), (3,c), (4,d), (5,e) \}
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.

\[\forall n \in \text{dom ddict}: \text{ddict}(n).\text{type} = \text{type}\? \Rightarrow \text{ddict}(n) \in \text{rng entries}!\]
\[\forall 1 \leq i \leq \#\text{entries}!: \text{entries}!(i).\text{type} = \text{type}\?\]
\[\forall 1 \leq i \leq \#\text{entries}!: \text{entries}!(i) \in \text{rng ddict}\]
\[\forall i,j \in \text{dom entries}!: (i < j) \Rightarrow \text{entries}!(i).\text{name} <_{\text{name}} \text{entries}!(j).\text{name}\]
We need invariants to define the range and the mapping.
Extract predicate

- For all entries in the data dictionary whose type is *type?*, there is an entry in the output sequence
- The type of all members of the output sequence is *type?*
- All members of the output sequence are members of the range of *ddict*
- The output sequence is ordered by entry name
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.

Objectives

• What is the role of formal software specification in the software process?

• What are the pros and cons of formal specification?

• When can formal specification be cost effective?
Software Process

• As specification is developed in detail, understanding of the specification increases. Creating a formal specification usually reveals errors in the informal specification, that are fed back to allow correction.

Why no formal specifications?

• Management is unwilling to adopt techniques without an obvious payoff.

• Software engineers lack training in discrete math and logic.

• Customers are unfamiliar with formal techniques.

• Certain classes of software systems are difficult to specify using existing techniques.

• Practicality of techniques is not well known.

• Little effort has been devoted to method and tool support for formal techniques.
Why no formal specifications?

(2)

• The move to interactive systems
  – Most formal specification languages are unable to cope with interactive graphical interfaces.
  – Difficult to integrate formal methods with rapid prototyping.

• Successful software engineering
  – If major progress is being made in improving software quality without using formal methods, why should they be introduced now?

Pros of formal specification

• Provides insight into understanding requirements and the software design.
  – Reduces requirement errors.

• May be analyzed using mathematical methods.
  – Ability to reason about the specification

• May be automatically processed.
  – Development and debugging tools

• May guide testing.
  – Identifying appropriate test cases
Transformational development

- Easier to prove correctness incrementally:

Myths of formal methods (Hall 1990)

- Formal methods result in perfect software
- Formal methods mean program proving
- Formal methods increase development costs
- Formal methods require a high level of mathematical skill
- Customers cannot understand formal specifications
- Formal methods have only been used for trivial system development
Development costs

Verdict on formal specification

- For a large class of systems, particularly highly interactive systems, formal specification is probably not cost effective for the foreseeable future.
- Where system dependability (safety, reliability, security, etc.) is critical, formal methods will probably become common.
- Formal specifications may also be very useful in defining standards where precision and unambiguity are essential.