Formal Specification

- Techniques for the unambiguous specification of software

Specification in the Software Process

- Specification and design are intermingled.
- Architectural design is essential to structure a specification.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and Design

- Increasing contractor involvement
- Decreasing client involvement
- Specification
- Design
- Architectural design
- Software specification
- High-level design
- Requirements definition
- Requirements specification
Formal Specification on Trial

- Formal techniques are not widely used in industrial software development.
- Given the relevance of mathematics in other engineering disciplines, why is this the case?

Why Aren’t Formal Methods Used?

- Inherent management conservatism. It is hard to demonstrate the advantages of formal specification in an objective way.
- Many software engineers lack the training in discrete math necessary for formal specification.
- System customers may be unwilling to fund specification activities.
- Some classes of software (particularly interactive systems and concurrent systems) are difficult to specify using current techniques.

Why Aren’t Formal Methods Used?

- There is widespread ignorance of the applicability of formal specifications.
- There is little tool support available for formal notations.
- Some computer scientists who are familiar with formal methods lack knowledge of the real-world problems to which these may be applied and therefore oversell the technique.
Advantages of Formal Specification

- It provides insights into the software requirements and the design.
- Formal specifications may be analyzed mathematically for consistency.
- It may be possible to prove that the implementation satisfies the specification.

Advantages of Formal Specification

- Formal specifications may be used to guide the tester of the component in identifying appropriate test cases.
- Formal specifications may be processed using software tools. It may be possible to animate the specification to provide a software prototype.

Seven Myths of Formal Methods

- Perfect software results from formal methods
  - Nonsense - the formal specification is a model of the real-world and may incorporate misunderstandings, errors and omissions.
- Formal methods means program proving
  - Formally specifying a system is valuable without formal program verification as it forces a detailed analysis early in the development process.
- Formal methods can only be justified for safety-critical systems.
  - Industrial experience suggests that the development costs for all classes of system are reduced by using formal specification.
Seven Myths of Formal Methods

- Formal methods are for mathematicians
  - Nonsense - only simple math is needed.
- Formal methods increase development costs
  - Not proven. However, formal methods definitely push development costs towards the front-end of the life cycle.
- Clients cannot understand formal specifications
  - They can if they are paraphrased in natural language.
- Formal methods have only been used for trivial systems
  - There are now many published examples of experience with formal methods for non-trivial software systems.

The Verdict!

- The reasons put forward for not using formal specifications and methods are weak.
- However, there are good reasons why these methods are not used:
  - The move to interactive systems. Formal specification techniques cannot cope effectively with graphical user interface specification.
  - Successful software engineering. Investing in other software engineering techniques may be more cost-effective.

Use of Formal Methods

- These methods are unlikely to be widely used in the foreseeable future. Nor are they likely to be cost-effective for most classes of system.
- They will become the normal approach to the development of safety critical systems and standards.
- This changes the expenditure profile through the software process.
Development Costs with Formal Specification

Specifying Functional Abstractions

- The simplest specification is function specification. There is no need to be concerned with global state.
- The formal specification is expressed as input and output predicates (pre and post conditions).
- Predicates are logical expressions which are always either true or false.
- Predicate operators include the usual logical operators and quantifiers such as for-all and exists.

Specification with Pre and Post Conditions

- Set out the pre-conditions
  - A statement about the function parameters stating what is invariably true before the function is executed
- Set out the post-conditions
  - A statement about the function parameters stating what is invariably true after the function has executed
- The difference between the pre and post conditions is due to the application of the function to its parameters. Together the pre and post conditions are a function specification.
Specification Development

- Establish the bounds of the input parameters. Specify this as a predicate.
- Specify a predicate defining the condition which must hold on the result of the function if it computes correctly.
- Establish what changes are made to the input parameters by the function and specify these as a predicate.
- Combine the predicates into pre and post conditions.

The Specification of a Search

```
function Search (X: array 1 .. N of INTEGER; Key: INTEGER )
return INTEGER ;
Pre: 3 i \leq N \cdot X(i) = Key
Post: X' (Search (X, Key)) = Key \land X = X'
```

Formal Specification Approaches

- Algebraic approach
  - The system is described in terms of interface operations and their relationships.
- Model-based approach
  - A model of the system acts as a specification. This model is constructed using well-understood mathematical entities such as sets and sequences.
Formal Specification Languages

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic</td>
<td>Larch (Guttag et al., 1985), LOTOS (Borgström and Brinksma, 1987)</td>
<td>Larch (Guttag et al., 1985)</td>
</tr>
<tr>
<td>Model-based</td>
<td>Z (Spivey, 1989)</td>
<td>VDM (Jones, 1980)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CSP (Hoare, 1985)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Petri Nets (Peterson, 1981)</td>
</tr>
</tbody>
</table>

Set Theory Review

\[ O = P - (T \cup S) \]

Implication

\[ a \Rightarrow b \equiv \neg a \lor b \]

\[ a = \text{possibility of precipitation (p.o.p)} > 50\% \]
\[ b = \text{take an umbrella} \]

So \( \Rightarrow \) informally means if the p.o.p is greater than 50\% I will take an umbrella, but if the p.o.p is less than or equal to 50\%, I may or may not take an umbrella.
Pre/Post Condition Examples

proc Reverse(a:array 0..9 of int, size:int)
pre: ?
post: ?

proc D_Sort(A:array 0..9 of int, size:int)
pre: ?
post: ?

Pre/Post Condition Examples(2)

proc Avg(A:array 0..9 of int, size:int, res:real)
pre: ?
post: ?

proc Init(A:array 0..9 of int, size:int)
pre: ?
post: ?

Pre/Post Condition Examples(3)

proc m_mult(A:array 0..9,0..9 of int; a1, a2: int;
B:array 0..9,0..9 of int; b1, b2: int;
M:array 0..9,0..9 of int; m1, m2: int)
pre: ?
post: ?

\[
\begin{align*}
\sum_{i=1}^{n} & A_i \cdot B_i \\
M & = A \cdot B + A \cdot B
\end{align*}
\]
Pre/Post Condition Examples (4)

process `count` defined on an array `A` of integers, where

- **pre:** ?
- **post:** ?

Number of `e`'s in `A`.

Pre/Post Conditions with Java (I)

```java
import java.util.*;

class Stack {
    private Vector v;

    void push(Object obj) {
        pre: (obj != null) ∧ (v != null)
        post: (v.addElementAt(v.size()) = obj)
        ∧ (0 ≤ i < v.size())
        ∧ v.elementAt(i) = v.elementAt(i)
        ∧ v.size() = v.size() + 1
    }

    Object pop() {
        pre: v != null
        post: (v.size > 0) ⇒ (v' = v.elementAtAt(v.size()-1))
        ∧ (v.size < 0) ⇒ (v' = null)
        ∧ (0 ≤ i < v.size()-1) ⇒ (v'.elementAt(i) = v.elementAt(i))
        ∧ v.size() = v.size()-1
    }
}
```

Pre/Post Conditions with Java (II)

```java
Object top() {
    pre: v != null
    post: (v.size ≤ 0) ⇒ (top' = null)
    ∧ (v.size > 0) ⇒ (top' = v.elementAtAt(v.size()-1))
    ∧ (v.size < 0) ⇒ (top' = null)
    ∧ (v.size ≥ 0) ⇒ (v.size() = v.size())
}

stack() {
    pre: true
    post: (v != null) ∧ (v.size() = 0)
}
```
Stack Implementation in Java

```java
import java.util.*;

class stack {
    private Vector v;
    void push(Object obj) {
        v.addElement(obj);
    }
    Object pop() {
        if (v.size() > 0) {
            Object res = v.elementAt(v.size()-1);
            v.removeElementAt(v.size()-1);
            return res;
        } else return null;
    }
    Object top() {
        if (v.size > 0)
            return v.elementAt(v.size()-1);
        else return null;
    }
    Stack() {
        v = new Vector();
        v.removeAllElements();
    }
}
```

Pre/Post Conditions for C functions

strlen()

In order to be able to work with C strings, we can introduce function:
- `isNullTerminated(s)`: returns True if the string s is null terminated and False otherwise.

```
int strlen(const char *s)
pre: isNullTerminated(s)
post: (s' = s) \land \exists 0 \leq i \cdot (s[i] = \0 \land (\forall 0 \leq j < i \cdot (s[j] \neq \0 ))) \implies strlen(s) = i
```

strchr() solution

```
char *strchr(const char *s, int c)
pre: ?
post: ?
```
Pre/Post Conditions for C functions

strcmp()

int strcmp(const char *s1, const char *s2)

// The strcmp() function compares two strings byte-by-byte, according
// to the ordering of your machine's character set. The function returns an
// integer greater than, equal to, or less than 0, if the string pointed to
// by s1 is greater than, equal to, or less than the string pointed to by s2
// respectively. The sign of a non-zero return value is determined by the
// sign of the difference between the values of the first pair of bytes that
// differ in the strings being compared. Bytes following a null byte are not
// compared.

int *strcmp(const char *s1, const char *s2)

pre: ?
post: ?

Model-based Specification

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

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

```
contents <= capacity
```

```
Container
contents: capacity:
```

```
contents <= capacity
```
An Indicator Specification

light = on ⇔ reading ≥ danger_level

Indicator
light: {off, on}
reading:
danger_level:
light = on ⇔ reading ≥ danger_level

Storage Tank Specification

storage_tank

storage_tank
Container
Indicator
reading = contents
capacity = 5000
danger_level = 50

Full Specification of a Storage Tank

storage_tank

storage_tank
contents:
capacity:
reading:
danger_level:
light: {off, on}
contents ≥ capacity
light = on ⇔ reading ≥ 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 Specification of a Fill Operation

- Fill-OK
  - \[ \Delta \text{Storage_tank} \]
  - amount?: \[ \text{contents} + \text{amount} \leq \text{capacity} \]
  - contents' = contents + amount?

Storage Tank Fill Operation

- OverFill
  - \[ \exists \text{Storage-tank} \]
    - amount?: \[ \text{contents} + \text{amount} < \text{capacity} \]
    - rl: seq CHAR
    - rl = "Insufficient tank capacity – Fill cancelled"

- Fill
  - Fill-OK \lor OverFill

The Z Specification Process

- Define given sets and types
- Define state variables
- Define initial state
- Define correct operations
- Define exceptional operations
- Combine operation schemas
- Write informal specification
- Decompose system
- Specify system components
- Compose component specifications
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
  - 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.
  - \( \text{SmallSquare} = \{1 \rightarrow 1, 2 \rightarrow 4, 3 \rightarrow 9, 4 \rightarrow 16, 5 \rightarrow 25, 6 \rightarrow 36, 7 \rightarrow 49 \} \)
- The domain of a function is the set of inputs over which the function has a defined result.
  - \( \text{dom SmallSquare} = \{1, 2, 3, 4, 5, 6, 7 \} \)
- The range of a function is the set of results which the function can produce.
  - \( \text{rng SmallSquare} = \{1, 4, 9, 16, 25, 36, 49 \} \)

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**

Init-DataDictionary

Δ DataDictionary

ddict' = Ø
Add and Lookup Operations

Add_OK

\[ \Delta \text{DataDictionary} \]
name?: NAME
entry?: DataDictionaryEntry

name? \not\in \text{dom ddict}

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

Lookup_OK

\[ \Xi \text{DataDictionary} \]
name?: NAME
entry!: DataDictionaryEntry

name? \in \text{dom ddict}

entry! = ddict (name?)

Add_Error

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

name? \not\in \text{dom ddict}

error! = "Name not in dictionary"

Lookup_Error

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

Function Overriding 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

\[ \Delta \text{DataDictionary} \]

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

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

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

\[ \text{ddict}' \oplus \{\text{name?} \mapsto \text{entry?}\} \]

---

Deleting an Entry

- Uses the domain subtraction operator which, given a name, removes that name from the domain of the function.

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

\[ \{\text{Ian}\} \ \text{phone} \]

\[ \{\text{Ray} \mapsto 3392, \text{Steve} \mapsto 3427\} \]

---

Delete Entry

\[ \Delta \text{DataDictionary} \]

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

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

\[ \text{ddict}' = \{\text{name?}\} \bowtie \text{ddict} \]
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 repl!: seq DataDictionaryEntry
in_type?: Sem_model_types

\[ \forall n : \text{dom } \text{ddict} \cdot \text{ddict}(n). \text{type} = \text{in_type} \Rightarrow \text{ddict}(n) \in \text{rng repl!} \]

\[ \forall i : 1 \leq i \leq \#\text{rep!} \cdot \text{rep!}(i).\text{type} = \text{in_type} \]

\[ \forall i, j : \text{dom } \text{rep!} \cdot (i < j) \Rightarrow \text{rep!}(i) < \text{name}(j) \text{ rep!}.\text{name}(j) \]

Extract Predicate

- For all entries in the data dictionary whose type is \text{in_type}, there is an entry in the output sequence.
- The type of all members of the output sequence is \text{in_type}.
- All members of the output sequence are members of the range of \text{ddict}.
- The output sequence is ordered.

Data Dictionary Specification

The\_Data\_Dictionary

DataDictionary
Init-DataDictionary
Add
Lookup
Delete\_OK
Replace\_OK
Extract
**Z Examples**

**Arrivals**

- Car? : CAR
- `p?` : pump
  - `p?` ∈ dom queues
  - car? ≠ head queues(p?)
  - queues(p?) = queues(0) \∩ tail queues(p?)
  - serviced = serviced \cup [car?]

**Switches**

- Car? : CAR
- `p?` : pump
  - `p?` ∈ dom queues \& `q?` ∈ dom queues
  - car? ∈ rng queues(p?)
  - queues(p?) = queues(p?) \cap queues(q?) \cap queues(p?) \cap queues(q?)

**LeavesUnhappy**

- In = {}
- Out = {}
- Users = {}

**Init State**

- New? : STAFF_ID
- Message! : REPORT
  - New? ∈ Users
  - Message! = AlreadyUser
  - Message! = Success

**AAHOK**

- Old State
- New State
- New? : STAFF_ID
- Message! : REPORT
- AlreadyUser
- Success
- Message! = OK
Z Examples (4)

CheckIn:
- Staff?: STAFF_ID
  - Staff? ∈ in → in' = in \ {Staff?} 
  - out' = out \ {Staff?} 
  - users' = users

NoUser:
- Staff?, Staff?: STAFF_ID 
  - message! : REPORT 
  - Staff? ∈ users → message! = "Already in"
  - Staff? ∉ users → message! = "Not a user"

Symbol Sheet

- State Variables change: ∆
- There exists at least one: ∃
- No state change occurs for the named state variable: ∆
- Equivalent: ≡
- Such That
- •
- Equivalent: ≡
- True if either a or b is true: OR
- True if both a and b are true: AND
- a if and only if b: ⇔
- if a then b: ⇒
- Summation: ∑
- For all elements in a set: ∀