Chapter 10
Formal Specification
Objectives

- To explain why formal specification helps discover problems in system requirements.
- To describe the use of:
  - **Algebraic** specification techniques, and
  - **Model-based** specification techniques (including simple pre- and post-conditions).
Formal methods

- Formal specification is part of a more general collection of techniques known as “formal methods.”
- All are based on the mathematical representations and analysis of requirements and software.
Formal methods (cont’d)

- Formal methods include:
  - Formal specification
  - Specification analysis and property proofs
  - Transformational development
  - Program verification (program correctness proofs) (axiomatic, function theoretic)

- Specifications are expressed with precisely defined vocabulary, syntax, and semantics.
Acceptance and use

- Formal methods have not become main-stream as was once predicted, especially in the US. Some reasons why:

  1. Less costly techniques (e.g., inspections / reviews) have been successful at increasing system quality. (Hence, the need for formal methods has been reduced.)

(Cont’d)
Acceptance and use (cont’d)

2. **Market changes** have made *time-to-market* rather than *quality* the key issue for many systems. *(Formal methods do not reduce time-to-market.)*

3. **Limited scope of formal methods.** They’re not well-suited to specifying *user interfaces*. *(Many interactive applications are “GUI-heavy” today.)*

(Cont’d)
Acceptance and use (cont’d)

4. Formal methods are hard to scale up for very large systems. (Although this is rarely necessary.)
5. Start-up costs are high.
6. Thus, the risks of adopting formal methods on most projects outweigh the benefits.
Acceptance and use (cont’d)

- However, formal specification is an excellent way to find (at least some types of) requirements errors and to express requirements unambiguously.

- Projects which use formal methods invariably report fewer errors in the delivered software.
Acceptance and use (cont’d)

- In systems where **failure must be avoided**, the use of formal methods is **justified** and likely to be **cost-effective**.

- Thus, the **use** of formal methods **is increasing in critical system development** where **safety, reliability, and security** are important.
Formal specification in the software process

- Requirements elicitation
- System modelling
- Architectural design
- High-level design
- Formal specification

A “back-end” element of requirements elicitation/analysis/specification/validation
Formal specification techniques

- **Algebraic approach** – system is specified in terms of its operations and their relationships via **axioms**.

- **Model-based approach** (including simple pre- and post-conditions) – system is specified in terms of a **state model** and operations are defined in terms of **changes to system state**.
Formal specification languages

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algebraic</strong></td>
<td>Larch (Guttag, Horning et al., 1985; Guttag, Horning et al., 1993), OBJ (Futatsugi, Goguen et al., 1985)</td>
<td>Lotos (Bolognesi and Brinksma, 1987),</td>
</tr>
<tr>
<td><strong>Model-based</strong></td>
<td>Z (Spivey, 1992)</td>
<td>CSP (Hoare, 1985)</td>
</tr>
<tr>
<td></td>
<td>VDM (Jones, 1980)</td>
<td>Petri Nets (Peterson, 1981)</td>
</tr>
<tr>
<td></td>
<td>B (Wordsworth, 1996)</td>
<td></td>
</tr>
</tbody>
</table>
Use of formal specification

- Formal specification is a *rigorous process* and *requires more effort in the early phases* of software development.
- This reduces requirements errors as *ambiguities, incompleteness, and inconsistencies* are discovered and resolved.
- Hence, *rework due to requirements problems* is greatly reduced.
Development costs with formal specification

- Without formal specification:
  - Specification
  - Design and Implementation
  - Validation

- With formal specification:
  - Specification
  - Design and Implementation
  - Validation
Algebraic Specification of sub-system interfaces

- Large systems are normally comprised of sub-systems with well-defined interfaces.
- Specification of these interfaces allows for their independent development.
- Interfaces are often defined as abstract data types ("sorts") or objects.
- The algebraic approach is particularly well-suited to the specification of such interfaces.
Sub-system interfaces

Sub-system A

Sub-system B

Interface objects
The structure of an algebraic specification

- `<SPECIFICATION NAME>` (Generic Parameter)
  - `sort <name>`
  - `imports <LIST OF SPECIFICATION NAMES>`

- Informal description of the sort and its operations

- Operation signatures setting out the names and the types of the parameters to the operations defined over the sort

- Axioms defining the operations over the sort
Algebraic specification components

- **Introduction** – defines the sort (type name) and declares other specifications that are used
- **Description** – informally describes the operations on the type
- **Signature** – defines the syntax of the operations in the interface and their parameters
- **Axioms** – defines the operation semantics by defining axioms which characterize behavior
Types of operations

- **Constructor operations**: operations which create / modify entities of the type
- **Inspection operations**: operations which evaluate entities of the type being specified
- **Rule of thumb** for defining axioms: define an axiom which sets out what is always true for each inspection operation over each (primitive) constructor operation.
Operations on a (FIFO linear) “List” abstract data type

- **Constructor operations** which create or modify sort List: Create, Cons and Tail
- **Inspection operations** which discover attributes of sort List: Head and Length
  (LISP fans: Tail = CDR, Head = CAR)
- Tail is not a **primitive** operation since it can be defined using Create and Cons. (Thus, axioms are not required for Head and Length over Tail.)
**List specification**

A FIFO linear list

Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence; Cons, which creates a new list with an added member; Length, which evaluates the list size; Head, which evaluates the front element of the list; and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type Elem.

<table>
<thead>
<tr>
<th>List operations</th>
<th>Type info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create → List</td>
<td></td>
</tr>
<tr>
<td>Cons (List, Elem) → List</td>
<td></td>
</tr>
<tr>
<td>Head (List) → Elem</td>
<td></td>
</tr>
<tr>
<td>Length (List) → Integer</td>
<td></td>
</tr>
<tr>
<td>Tail (List) → List</td>
<td></td>
</tr>
</tbody>
</table>

1. Head (Create) = Undefined exception (empty list)
2. Head (Cons (L, v)) = if L=Create then v else Head (L)
3. Length (Create) = 0
4. Length (Cons (L, v)) = Length (L) + 1
5. Tail (Create) = Create
6. Tail (Cons (L, v)) = if L=Create then Create else Cons (Tail (L), v)

Defines Tail in terms of Create and Cons

---

© Ian Sommerville 2000

Software Engineering, Chapter 10

Slide 21
Recursion in specifications

- \( \text{Tail} (\text{Cons} (L, v)) = \begin{cases} \text{Create} & \text{if } L = \text{Create} \\ \text{Cons} (\text{Tail} (L), v) & \text{else} \end{cases} \)

\[ \begin{align*}
\text{Tail} (\text{Cons} ([5, 7], 9)) &= ? \\
&= \text{Cons} (\text{Tail} ([5, 7]), 9) \quad (\text{axiom 6}) \\
&= \text{Cons} (\text{Tail} (\text{Cons} ([5], 7)), 9) \\
&= \text{Cons} (\text{Cons} (\text{Tail} ([5]), 7), 9) \quad (\text{axiom 6}) \\
&= \text{Cons} (\text{Cons} (\text{Tail} (\text{Cons} ([], 5)), 7), 9) \\
&= \text{Cons} (\text{Cons} (\text{Create}, 7), 9) \quad (\text{axiom 6}) \\
&= \text{Cons} ([7], 9) \\
&= [7, 9]
\]
Exercise

What does Head (Tail (L)) do?

<table>
<thead>
<tr>
<th>L</th>
<th>Head (Tail (L))</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>undefined</td>
</tr>
<tr>
<td>[a]</td>
<td>undefined</td>
</tr>
<tr>
<td>[a, b]</td>
<td>b</td>
</tr>
<tr>
<td>[a, b, c]</td>
<td>b</td>
</tr>
</tbody>
</table>
Exercise

Are axioms 1-6 sufficient to prove ANY true assertion of the form

\[ \text{Head (Tail (L) )} = v \]

Consider ONE EXAMPLE:

\[ \text{Head (Tail ([a, b])} = b \]
Proof that Head (Tail ([a, b])) = b

\[
\text{Head (Tail ([a, b]))} = \text{Head (Tail (\text{Cons ([a], b)}))} \\
= \text{Head (\text{Cons (Tail ([a]), b)})} \quad \text{(axiom 6)} \\
= \text{Head (\text{Cons (Tail (\text{Cons ([], a)}, b))})} \\
= \text{Head (\text{Cons ([], b)})} \quad \text{(axiom 6)} \\
= \text{b} \quad \text{(axiom 2)}
\]
Question

So, we can prove Head (Tail ([a, b]) = b using the given axioms, but how could one show that the axioms are sufficient to prove ANY true assertion of the form

\[ \text{Head (Tail (L) ) = v ?} \]

Moral: Showing correctness and completeness of algebraic specifications can be very tricky!
Model-based specification

- Algebraic specification can be cumbersome when object operations are not independent of object state (i.e., the result of previous operations).
- (System State) Model-based specification exposes the system state and defines operations in terms of changes to that state.
Model-based specification

- Z is a mature notation for model-based specification. It combines formal and informal descriptions and incorporates graphical highlighting.
- The basic building blocks of Z-based specifications are schemas.
- Schemas identify state variables and define constraints and operations in terms of those variables.
The structure of a Z schema

Container

- contents: \( \mathbb{N} \)
- capacity: \( \mathbb{N} \)

- contents \( \leq \) capacity

N
Natural numbers (0, 1, 2, …)

(defines scheme state)

(invariants, pre- & post-conditions)
An insulin pump

- Insulin reservoir
- Insulin delivery
- Glucose level
- Text messages
- Dose delivered
- Needle assembly
- Pump
- Controller
- Alarm
- Clock
- Display 1
- Display 2
- Power supply
- Insulin reservoir
Modelling the insulin pump

- The schema models the insulin pump as a number of state variables
  - reading? from glucose level sensor
  - dose, cumulative_dose
  - r0, r1, r2 last 3 glucose level readings
  - capacity insulin pump reservoir level
  - alarm! signals exceptional conditions
  - pump! output for insulin pump device
  - display1!, display2! text messages & dose

- Names followed by a ? are inputs, names followed by a ! are outputs.
Insulin pump schema signature

INSULIN_PUMP_STATE

//Input device definition

switch?: (off, manual, auto)
ManualDeliveryButton?:  
Reading?:  
HardwareTest?: (OK, batterylow, pumpfail, sensorfail, deliveryfail)
InsulinReservoir?: (present, notpresent)
Needle?: (present, notpresent)
clock?: TIME

//Output device definition

alarm! = (on, off)
display1!, string
display2!: string
clock!: TIME
dose!:  

©Ian Sommerville 2000  Software Engineering, Chapter 10
Insulin pump schema **signature** (cont'd)

```c
// INSULIN_PUMP_STATE  (Cont'd)

// State variables used for dose computation

status: (running, warning, error)
r0, r1, r2: N
capacity, insulin_available : N
max_daily_dose, max_single_dose, minimum_dose: N
safemin, safemax: N
CompDose, cumulative_dose: N
```
Schema predicates

- Each Z schema has an predicate part which defines conditions that are always true (schema invariants)

- For the insulin pump schema, for example, it is always true that:
  - The dose must be less than or equal to the capacity (= level) of the insulin reservoir.
  - No single dose may be more than 4 units of insulin and the total dose delivered in a time period must not exceed 25 units of insulin. This is a safety constraint.
  - display2! shows the amount of insulin to be delivered.
// INSULIN_PUMP_STATE (Cont'd)

r2 = Reading?
dose! ≤ insulin_available
insulin_available ≤ capacity

// The cumulative dose of insulin delivered is set to zero once every 24 hours
clock? = 000000 ⇒ cumulative_dose = 0

// If the cumulative dose exceeds the limit then operation is suspended
cumulative_dose ≥ max_daily_dose ∧ status = error ∧
display1! = “Daily dose exceeded”

// Pump configuration parameters
capacity = 100 ∧ safemin = 6 ∧ safemax = 14
max_daily_dose = 25 ∧ max_single_dose = 4 ∧ minimum_dose = 1

display2! = nat_to_string (dose!)
clock! = clock?
The dosage computation

- The insulin pump computes the amount of insulin required by comparing the current reading with two previous readings.
- If these suggest that blood glucose is rising then insulin is delivered.
- Information about the total dose delivered is maintained to allow the safety check invariant to be applied.
- Note that this invariant always applies - there is no need to repeat it in the dosage computation.
RUN schema

- operations change state
- imports state & predicates

RUN

\[ \Delta \text{INSULIN\_PUMP\_STATE} \]

switch? = auto
status = running ∨ status = warning
insulin_available ≥ max_single_dose
cumulative_dose < max_daily_dose
// The dose of insulin is computed depending on the blood sugar level
(SUGAR\_LOW ∨ SUGAR\_OK ∨ SUGAR\_HIGH)
// 1. If the computed insulin dose is zero, don’t deliver any insulin
CompDose = 0 ⇒ dose! = 0
∨
// 2. The maximum daily dose would be exceeded if the computed dose was delivered so the insulin
dose is set to the difference between the maximum allowed daily dose and the cumulative dose
delivered so far
CompDose + cumulative_dose > max_daily_dose ⇒ alarm! = on ∧ status' = warning ∧ dose! = max_daily_dose – cumulative_dose
∨
x' - value of x after operation
RUN schema (cont'd)

// RUN (cont'd)

// 3. The normal situation. If maximum single dose is not exceeded then deliver the computed dose. If the single dose computed is too high, restrict the dose delivered to the maximum single dose

CompDose + cumulative_dose < max_daily_dose ⇒
   ( CompDose ≤ max_single_dose ⇒ dose! = CompDose
   \)
   CompDose > max_single_dose ⇒ dose! = max_single_dose

insulin_available' = insulin_available – dose!
cumulative_dose' = cumulative_dose + dose!

insulin_available ≤ max_single_dose * 4 ⇒ status' = warning ∧
display1! = “Insulin low”

r1' = r2
r0' = r1
Sugar OK schema

\[
SUGAR\_OK
\]

\[
\begin{align*}
r2 & \geq \text{safemin} \land r2 \leq \text{safemax} \\
& // \text{sugar level stable or falling} \\
r2 & \leq r1 \implies \text{CompDose} = 0 \\
\lor \\
& // \text{sugar level increasing but rate of increase falling} \\
r2 & > r1 \land (r2-r1) < (r1-r0) \implies \text{CompDose} = 0 \\
\lor \\
& // \text{sugar level increasing and rate of increase increasing compute dose} \\
& // a minimum dose must be delivered if rounded to zero \\
r2 & > r1 \land (r2-r1) \geq (r1-r0) \land (\text{round } ((r2-r1)/4) = 0) \implies \\
& \quad \text{CompDose} = \text{minimum\_dose} \\
\lor \\
r2 & > r1 \land (r2-r1) \geq (r1-r0) \land (\text{round } ((r2-r1)/4) > 0) \implies \\
& \quad \text{CompDose} = \text{round } ((r2-r1)/4)
\end{align*}
\]
Specification via **Pre- and Post-Conditions**

- Predicates that (when considered together) reflect a program’s intended functional behavior are defined over its state variables.

- **Pre-condition**: expresses constraints on program variables before program execution. An implementer may assume these will hold BEFORE program execution.
Specification via Pre- and Post-Conditions (cont’d)

- **Post-condition**: expresses conditions / relationships among program variables after execution. These capture any obligatory conditions AFTER program execution.

- **Language**: *predicate calculus*
  - Predicates \((X>4)\)
  - Connectives \((&, V, \rightarrow, \leftrightarrow, \text{NOT})\)
  - Universal and existential quantifiers (“for every…”, “there exists…”)
  - Rules of inference (if \(A \& (A \rightarrow B)\) then \(B\))
Example 1

Sort a *non-empty* array \( \text{LIST}[1..N] \) into increasing order.

**Pre-cond:** \( N \geq 1 \)

**Post-cond:** For _Every_ \( i \), \( 1 \leq i \leq N-1 \), \( \text{LIST}[i] \leq \text{LIST}[i+1] \)

& \( \text{PERM(\text{LIST},\text{LIST’})} \)
Example 2

Search a non-empty, ordered array LIST[1..N] for the value stored in KEY. If present, set FOUND to true and J to an index of LIST which corresponds to KEY. Otherwise, set FOUND to false.
Example 2

**Pre-cond:**  \( N \geq 1 \) & \[
\text{“(LIST is in increasing order”) V (“LIST is in decreasing order”)}\]

(Exercise: express the “ordered” predicates above FORMALLY.)

**Post-cond:** \[
\text{(FOUND & There Exists i, 1 \leq i \leq N | J=i & LIST[J]=Key) V (NOT FOUND & For Every i, 1 \leq i \leq N, LIST[i] \neq KEY)}\] & UNCH(LIST,KEY)}
Exercise

- See **PRE- AND POST-CONDITION SPECIFICATION EXERCISE** on course website
Key points

- Formal system specification complements informal specification techniques.
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification.
- Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system.

(Cont’d)
Key points (cont’d)

- Formal specification techniques are most applicable in the development of critical systems.
- Algebraic techniques are particularly suited to specifying interfaces of objects and abstract data types.
- In model-based specification, operations are defined in terms of changes to system state.
Chapter 10

Formal Specification