Lecture 11 – Formal Specifications
Objectives:

• To explain why formal specification techniques help discover problems in system requirements
• To describe the use of:
  • algebraic techniques (for interface specification) and
  • model-based techniques (for behavioural specification)
• To introduce Abstract State Machine Model (ASML)
Formal Methods

- **Formal specification** is part of a more general collection of techniques that are known as ‘formal methods’
  - COMP313 “Formal Methods”
  - These are all based on the mathematical representation and analysis of software

- **Formal methods** include
  - Formal specification
  - Specification analysis and proof
  - Transformational development
  - Program verification
Formal Methods in reality

• When software was first developed
  • It was done using assembly language
  • No OO, no high level languages
  • Limited understanding of software testing

• Modern software development
  • Many ways to make high quality software

• So
  • Mostly formal methods not used
  • The most acceptable techniques are approaches like programming by contract (e.g. Eiffel)
Acceptance of Formal Methods

- Formal methods have not become mainstream software development techniques as was once predicted.
  - Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced.
  - Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market.
  - The scope of formal methods is limited. They are not well-suited to specifying and analysing user interfaces and user interaction for example.
  - Formal methods are hard to scale up to large systems.
Use of Formal Methods

• Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems:
  • Air traffic control information systems,
  • Railway signalling systems
  • Spacecraft systems
  • Medical control systems

• In this area, the use of formal methods is most likely to be cost-effective
Background Reading


“Software development failures have reached staggering proportions: an estimated $81 billion was spent on cancelled software projects in 1995 and an estimated $100 billion in 1996.” [1]

“Experience shows that many of the most vexing problems in software development arise because any computer system is situated in a particular social context.” [1]
Specification in the Software Process

- **Specification** and design are inextricably mixed.
- **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

Requirements definition
Requirements specification
Architectural design
Software specification
High-level design

Specification
Design
Specification Techniques

• **Algebraic approach**
  • The system is specified in terms of its operations and their relationships

• **Model-based approach**
  • The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences.
  • Operations are defined by modifications to the system’s state
Formal Specification Languages

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic</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>Model-based</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>

ASML - Abstract State Machine Language

Yuri. Gurevich, Microsoft Research, 2001
Use of Formal Specification

- **Formal specification** involves investing more effort in the early phases of software development.

  This reduces **requirements errors** as it forces a detailed analysis of the requirements.

- **Incompleteness** and **inconsistencies** can be discovered and resolved.

  Hence, savings are made as the amount of rework due to requirements problems is reduced.
Formal Specification

- Critical systems development usually follows a software process based on the **waterfall model**.
- The system requirements and design are expressed in detail, reducing ambiguity, and carefully analysed and refined before implementation begins.
- A large benefit of formal specification is its ability to uncover potential problems and ambiguities in the requirements.

**Question:** Why does this mean the waterfall model is often used?
Interface Specification

- Large systems are decomposed into subsystems with well-defined interfaces between these subsystems
- Specification of subsystem interfaces allows independent development of the different subsystems
- Interfaces may be defined as abstract data types or object classes

The algebraic approach to formal specification is particularly well-suited to interface specification
Clear and unambiguous sub-system interface specifications reduce the chance of misunderstandings between a provider and user of a sub-system.

The algebraic approach to specification was originally developed for the definition of abstract data types.

This idea was then extended to model complete system specifications.
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
The Structure of an Algebraic Specification

- **Introduction** – Declares the sort (the type name) of the entity being specified, i.e., a set of objects with common characteristics. It also imports other specifications to use.
- **Description** – An informal description of the operations to aid understanding.
- **Signature** – Defines the syntax of the interface to the abstract data type (object), including their names, parameter list and return types.
- **Axioms** – Defines the semantics of the operations by defining axioms characterising the behaviour.
Systematic Algebraic Specification

- **Algebraic specifications** of a system may be developed in a systematic way:
  - Specification structuring;
  - Specification naming;
  - Operation selection;
  - Informal operation specification;
  - Syntax definition;
  - Axiom definition.
Specification Operations

- **Constructor operations.** Operations which create entities of the type being specified.
- **Inspection operations.** Operations which evaluate entities of the type being specified.
- To specify behaviour, define the inspector operations for each constructor operation.
Example: Operations on a List ADT

• Let us take an example of a “list” abstract data type.
• A list contains a sequence of elements of some type where elements may be added to the end and removed from the front (this is also called a queue, how does this differ from a stack?).
• We want operations to Create, Cons (create a new list with an added member), Head (to evaluate the first element), Length and Tail (which creates a list by removing the first (head) element).
Example: Operations on a List ADT

- Constructor operations which evaluate to sort List
  - Create, Cons and Tail.
- Inspection operations which take sort list as a parameter and return some other sort
  - Head and Length.
- Tail can be defined using the simpler constructors Create and Cons. No need to define Head and Length with Tail.
Example: List Specification

<table>
<thead>
<tr>
<th>List(Elem)</th>
<th>sort List</th>
<th>Imports INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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.

Create -> List
Cons(List, Elem) -> List
Head (List) -> Elem
Length (List) -> Integer
Tail (List) -> List

Head(Create) = Undefined **exception** (empty List)
Head(Cons(L,v)) = if L = Create then v else Head (L)
Length(Create) = 0
Length(Cons(L,v)) = Length (L) + 1
Tail(Create) = Create
Tail(Cons(L,v)) = if L = Create then Create else Cons(Tail(L), v)
Interface Specification in Critical Systems

- Let us consider another example: an *air traffic control system* where aircraft fly through managed sectors of airspace.
- Each *sector* may include a number of aircraft but, for safety reasons, *these must be separated*.
- In this example, a simple vertical separation of 300m is proposed.
- The system should warn the controller if aircraft are instructed to move so that the separation rule is breached.
A Sector Object

- Critical operations on an object representing a controlled sector are:
  - **Enter** - Add an aircraft to the controlled airspace;
  - **Leave** - Remove an aircraft from the controlled airspace;
  - **Move** - Move an aircraft from one height to another;
  - **Lookup** - Given an aircraft identifier, return its current height;
Primitive Operations

- It is sometimes necessary to introduce additional operations to simplify the specification.
- The other operations can then be defined using these more primitive operations.

- Primitive operations
  - **Create** - Bring an instance of a sector into existence;
  - **Put** - Add an aircraft without safety checks;
  - **In-space** - Determine if a given aircraft is in the sector;
  - **Occupied** - Given a height, determine if there is an aircraft within 300m of that height.
**Sector Specification (1)**

<table>
<thead>
<tr>
<th>SECTOR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>Sector</td>
</tr>
<tr>
<td>import</td>
<td>INTEGER, BOOLEAN</td>
</tr>
</tbody>
</table>

Enter - adds an aircraft to the sector if safety conditions are satisfied
Leave - removes an aircraft from the sector
Move - moves an aircraft from one height to another if safe to do so
Lookup - Finds the height of an aircraft in the sector

Create - creates an empty sector
Put - adds an aircraft to a sector with no constraint checks
In-space - checks if an aircraft is already in a sector
Occupied - checks if a specified height is available

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean
Enter \((S, CS, H) = \)
\[\text{if } \text{In-space}\ (S, CS) \text{ then } S \text{ exception (Aircraft already in sector)} \]
\[\text{elsif } \text{Occupied}\ (S, H) \text{ then } S \text{ exception (Height conflict)} \]
\[\text{else } \text{Put}\ (S, CS, H) \]

Leave \((\text{Create}, CS) = \text{Create exception (Aircraft not in sector)} \)
Leave \((\text{Put}\ (S, CS1, H1), CS) = \)
\[\text{if } CS = CS1 \text{ then } S \text{ else } \text{Put}\ (\text{Leave}\ (S, CS), CS1, H1) \]

Move \((S, CS, H) = \)
\[\text{if } S = \text{Create then Create exception (No aircraft in sector)} \]
\[\text{elsif } \text{not In-space}\ (S, CS) \text{ then } S \text{ exception (Aircraft not in sector)} \]
\[\text{elsif } \text{Occupied}\ (S, H) \text{ then } S \text{ exception (Height conflict)} \]
\[\text{else } \text{Put}\ (\text{Leave}\ (S, CS), CS, H) \]

\(-- \text{NO-HEIGHT is a constant indicating that a valid height cannot be returned}\ --\)

Lookup \((\text{Create}, CS) = \text{NO -HEIGHT exception (Aircraft not in sector)} \)
Lookup \((\text{Put}\ (S, CS1, H1), CS) = \)
\[\text{if } CS = CS1 \text{ then } H1 \text{ else } \text{Lookup}\ (S, CS) \]

Occupied \((\text{Create}, H) = \text{false} \)
Occupied \((\text{Put}\ (S, CS1, H1), H) = \)
\[\text{if } (H1 > H \text{ and } H1 - H \geq 300) \text{ or } (H > H1 \text{ and } H - H1 \geq 300) \text{ then true} \]
\[\text{else Occupied}\ (S, H) \]

In-space \((\text{Create}, CS) = \text{false} \)
In-space \((\text{Put}\ (S, CS1, H1), CS) = \)
\[\text{if } CS = CS1 \text{ then true else In-space}\ (S, CS) \]
Specification Commentary for Sector

- Use the basic constructors **Create** and **Put** to specify other operations.
- Define **Occupied** and **In-space** using **Create** and **Put** and use them to make checks in other operation definitions.
- All operations that result in changes to the sector must check that the safety criterion holds.
Lecture 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.
• Formal specification techniques are most applicable in the development of critical systems and standards.