Chapter 9

Formal Specification

Techniques for the unambiguous specification of software

Objectives

- To explain why formal specification techniques help discover problems in system requirements
- To describe the use of algebraic techniques for interface specification
- To describe the use of model-based techniques for behavioural specification

Topics covered

- Formal specification in the software process
- Interface specification
- Behavioural specification

Formal methods

- Formal specification is part of a more general collection of techniques that are known as 'formal methods'
- These are all based on logico-mathematical representation and analysis of software
- Formal methods include
  - Formal specification
  - Specification analysis and proof
  - Transformational development
  - Program verification

An unfulfilled prediction

It was predicted in the 1980s that, by the 21st century, a large proportion of software would be developed using formal methods.

The reasons why

- Other software engineering techniques have been successful at increasing system quality.
- Market changes have made time-to-market rather than software with a low error count the key factor.
- Formal methods are not well-suited for specifying and analysing user interfaces and user interaction.
- Formal methods are hard to scale up to large systems.

Usefulness of formal methods

- Their use often leads to fewer errors in systems.
- The main area of applicability is in critical systems, where its use is most likely to be cost-effective.

Specification in the software process

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

Specification and design

Specification in the software process

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

Languages for Formal specification

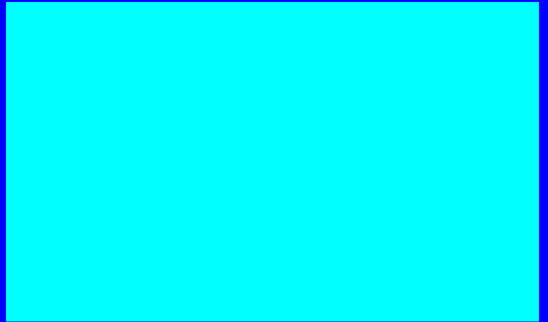
	Seqential programs	Concurrent programs
Algebraic	Larch (1993) OBJ (1985)	Lotos (1987)
Model-based	Z (1992) VDM (1980) B (1996)	CSP (1985) Petri Net (1981)

Use of formal specification

- It requires more effort in the early phases of software development.
- It reduces requirements errors because it entails a detailed analysis of the requirements.
- It often leads to earlier discovery and resolution of incompleteness and inconsistencies.
- Hence, savings are made as the amount of rework due to requirements problems is reduced.

Development costs with formal specification:

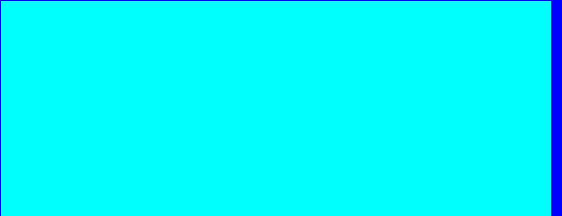
the key question is "what is the total cost?"



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.

Sub-system interfaces



Components of an algebraic spec.

- Introduction
  - Defines **`sort`** (the 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 characterise behaviour

Steps involved in developing an algebraic spec.

1. Construct an informal spec. and organize it into a set of abstract data types (ADT) or object
2. Name each ADT
3. Define a set of operations for each ADT
4. Write an informal specification for each operation
5. Define the syntax of each operation and its parameters
6. Define the semantics of the operations as a set of axioms

Specifying operations for an ADT

There should be

- Constructor operations: Operations for creating instances of the ADT, or for creating another from a given one.
- Inspection operations. Operations for accessing the content or attribute of an element.
- Define the inspection operations for each constructor operation (this determines the behavior of the ADT).

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.

List specification

LIST (Elem)

sort List
imports INTEGER

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.

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)

Recursion in specifications

- Operations are often specified recursively
- Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)
  - Cons ([5, 7], 9) = [5, 7, 9]
  - Tail ([5, 7, 9]) = Tail (Cons ([5, 7], 9)) =
  - Cons (Tail ([5, 7]), 9) = Cons (Tail (Cons ([5], 7)), 9) =
  - Cons (Cons (Tail ([5]), 7), 9) =
  - Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
  - Cons (Cons ([Create], 7), 9) = Cons ([7], 9) = [7, 9]

Interface specification in critical systems

- Consider 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

Specification commentary

- 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.

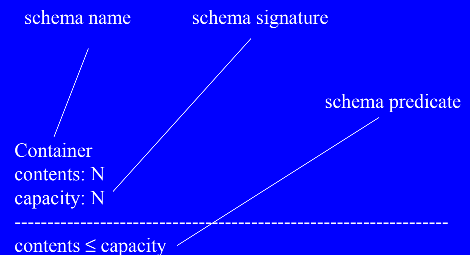
Behavioural specification

- Algebraic specification can be cumbersome when the object operations are not independent of the object state.
- Model-based specification exposes the system state and defines the operations in terms of changes to that state.
- The Z notation is a mature technique for model-based specification. It combines formal and informal description and uses graphical highlighting when presenting specifications.

Z schemas

- The formal description is included as small, easy to read chunks called schemas.
- They are used to introduce state variables and to define constraints and operations on the states.
- Schema operations include schema composition, schema renaming, and schema hiding.

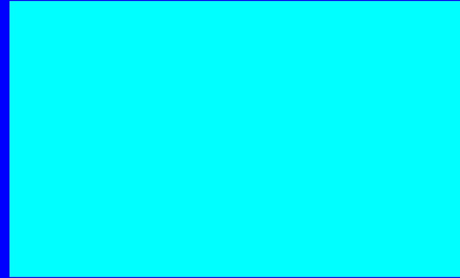
The structure of a Z schema



Z schema

- The schema signature defines the entities, the aggregate of which defines the system state.
- The schema predicate describes the condition that must be true.
- Where the schema is used to define an operation, the schema predicate may set out the pre- and post-conditions.

An insulin pump



Modelling the insulin pump

- The schema models the insulin pump as a number of state variables
  - reading?
  - dose, cumulative_dose
  - r0, r1, r2
  - capacity
  - alarm!
  - pump!
  - display1!, display2!
- Names followed by a ? are inputs, and by a ! outputs

Schema invariant

- Each Z schema has an invariant part which defines conditions that are always true
- For the insulin pump schema it is always true that
  - The dose must be less than or equal to the capacity of the insulin reservoir.
  - No single dose may be more than 5 units of insulin and the total dose delivered in a time period must not exceed 50 units of insulin. This is a safety constraint.
  - display1! shows the status of the insulin reservoir.

insulin_pump — Insulin pump schema

```
reading?: N
dose, cumulative_dose: N
r0, r1, r2: N    //used to record the last 3 readings taken
capacity: N
alarm: {off, on}
pump!: N
display1!, display2!: STRING
---------------------------------------------------------------------------
dose ≤ capacity ∧ dose ≤ 5 ∧ cumulative_dose ≤ 50
capacity ≥ 40 ⇒ display1! = " "
capacity ≤ 39 ∧ capacity ≥ 10 ⇒ display1! = "insulin low"
capacity ≤ 9 ⇒ alarm! = on ∧ display1! = "insulin very low"
r2 = reading?
```


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

Dosage schema

```
DOSAGE
Δinsulin_pump

(
dose = 0 ∧
        (
                ((r1 ≥ r0) ∧ (r2 = r1)) ∨
                ((r1 > r0) ∧ (r2 ≤ r1)) ∨
                ((r1 < r0) ∧ (r1 - r2) > (r0 - r1)))
        ) ∨
dose = 4 ∧      (
                ((r1 ≤ r0) ∧ (r2 = r1)) ∨
                ((r1 < r0) ∧ (r1 - r2) ≤ (r0 - r1)))
        ) ∨
dose = (r2 - r1) * 4 ∧
        (
                ((r1 ≤ r0) ∧ (r2 > r1)) ∨
                ((r1 > r0) ∧ (r2 - r1) ≥ (r1 - r0)))
        )
)
capacity' = capacity - dose
cumulative_dose' = cumulative_dose + dose
r0' = r1 ∧ r1' = r2
```


Output schemas

- The output schemas model the system displays and the alarm that indicates some potentially dangerous condition
- The output displays show the dose computed and a warning message
- The alarm is activated if blood sugar is very low - this indicates that the user should eat something to increase their blood sugar level

Output Schemata

```
DISPLAY
Δinsulin_pump

display2!' = Nat_to_string (dose) ∧
(reading? < 3 ⇒ display1!' = "sugar low" ∨
reading? > 30 ⇒ display1!' = "sugar high" ∨
reading? ≥ 3 ∧ reading? ≤ 30 ⇒ display1!' = "OK")
```

```
ALARM
Δinsulin_pump

(reading? < 3 ∨ reading? > 30) ⇒ alarm!' = on ∨
(readiing? ≥ 3 ∧ reading? ≤ 30) ⇒ alarm!' = off
```


Schema consistency

- It is important that schemas are consistent. Otherwise, a problem with the system requirements is indicated.
- The INSULIN_PUMP and DISPLAY schemas are inconsistent
  - display1! shows a warning message about the insulin reservoir (INSULIN_PUMP)
  - display1! Shows the state of the blood sugar (DISPLAY)
- This must be resolved before implementation of the system

Key points

Formal specification

- complements the informal specification,
- is precise and unambiguous,
- allows a rigorous analysis of the system requirements at an early stage, and
- is most suitable for development of critical system.

Key points (continued)

- Algebraic methods are suitable for system interface specification where the interface is defined as a set of object classes.
- Model-based methods model a system using sets and functions. That simplifies specifications of certain software system behaviours.