Algebraic Specification

Reference:
Software Engineering By Sommerville (8th Edition)
Chapter 10

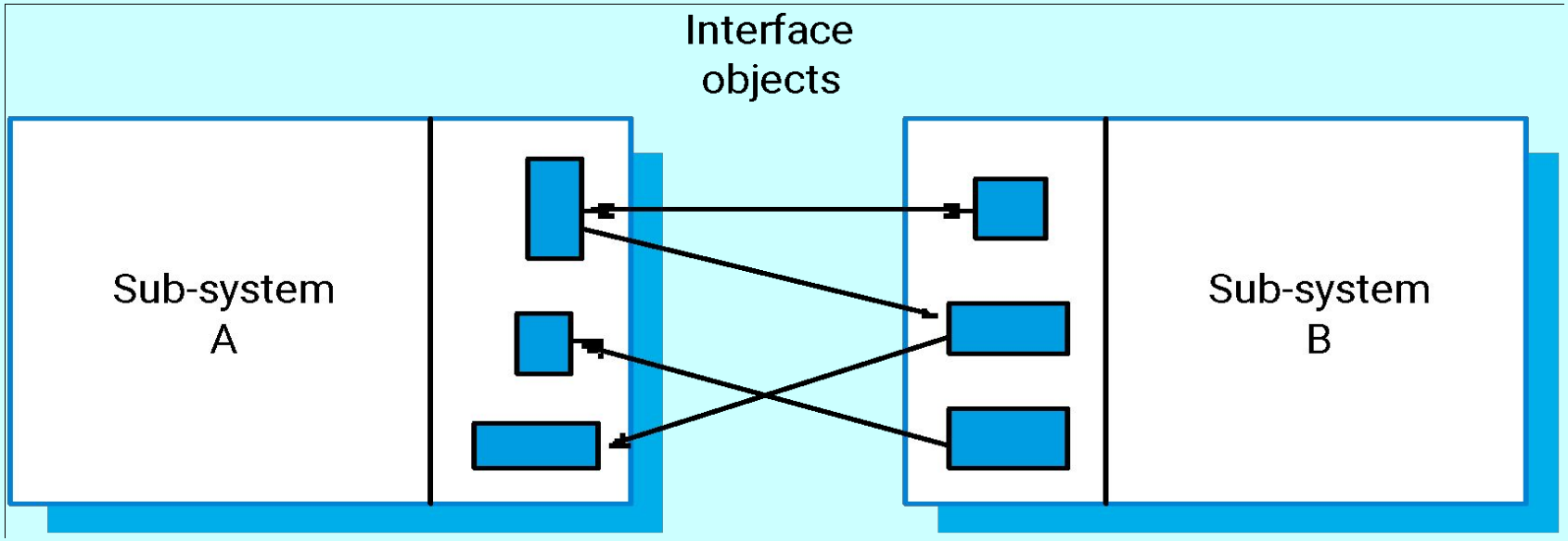
Types of specification

- Interface specification
  - How the subsystems are going to interact with each other
  - not about the details of particular subsystem
- Behavioural specification
  - Concerns about the behaviour of individual module in the system
  - Does not concern with intermodule relationships

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 as it is focused on the defined operations in an object.

Sub-system interfaces



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

Specification components

- Introduction
  - Defines the 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

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 inspection operations for each constructor operation

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

Primitive operations

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) =

If In-space(S,CS) then undefined exception(Aircraft already in sector)

else Occupied(S,H) then undefined exception(Height conflict)

else Put(S,CS,H)

Leave (Create, CS)

= Create undefined exception(aircraft not in sector)

Leave(Put (S,CS1,H1),CS)

=if CS=CS1 then S else Put(Leave (S,CS), CS1, H1)

Move (S,CS,H)

= if S=Create then Create Undefined Exception (No aircraft in sector)

else if not In-space(S, CS) then S undefined Exception(Aircraft not in sector)

else if Occupied(S,H) then S exception (Height conflict)

else Put(Leave(S,CS), CS, H)

Lookup(Create, CS) = NO-HEIGHT undefined exception(Aircraft not in sector)

Lookup(Put(S,CS1,H1), CS)

= if CS=CS1 then H1 else Lookup(S,CS)

Occupied(Create, H) = false

Occupied(Put(S, CS1,H1), H)

= if (H1> H and H1-H <=300) or (H > H1 and H-H1 >= 300) then true

else Occupied(S,H)

In-space(Create, CS)=false

In-space(Put(S,CS1, H1), CS)

= if CS=CS1 then true else In-space (S,CS)

SECTOR

sort Sector
imports INTEGER, BOOLEAN

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) =
  if In-space (S, CS) **then** S **exception** (Aircraft already in sector)
  elsif Occupied (S, H) **then** S **exception** (Height conflict)
  else Put (S, CS, H)

Leave (Create, CS) = Create **exception** (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
  if CS = CS1 **then** S **else** Put(Leave (S, CS), CS1, H1)

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

-- NO-HEIGHT is a constant indicating that a valid height cannot be returned

Lookup (Create, CS) = NO-HEIGHT **exception** (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
  if CS = CS1 **then** H1 **else** Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
  if (H1 > H **and** H1 - H Š 300) **or** (H > H1 **and** H - H1 Š 300) **then** true
  else Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS) =
  if CS = CS1 **then** true **else** In-space (S, CS)

A few ADTs to practice

Stack, Integer, Set of Integers, Bag of Integers,
text editor(refer Ghezzi book for text editor)