Algebraic specification

- ⊗ Specifying abstract types in terms of relationships between type operations

Objectives

- ⊗ To explain the role of formal specifications in sub-system interface definition
- ⊗ To introduce the algebraic approach to formal specification
- ⊗ To describe the systematic construction of algebraic specifications
- ⊗ To illustrate a number of incremental ways to write algebraic specifications

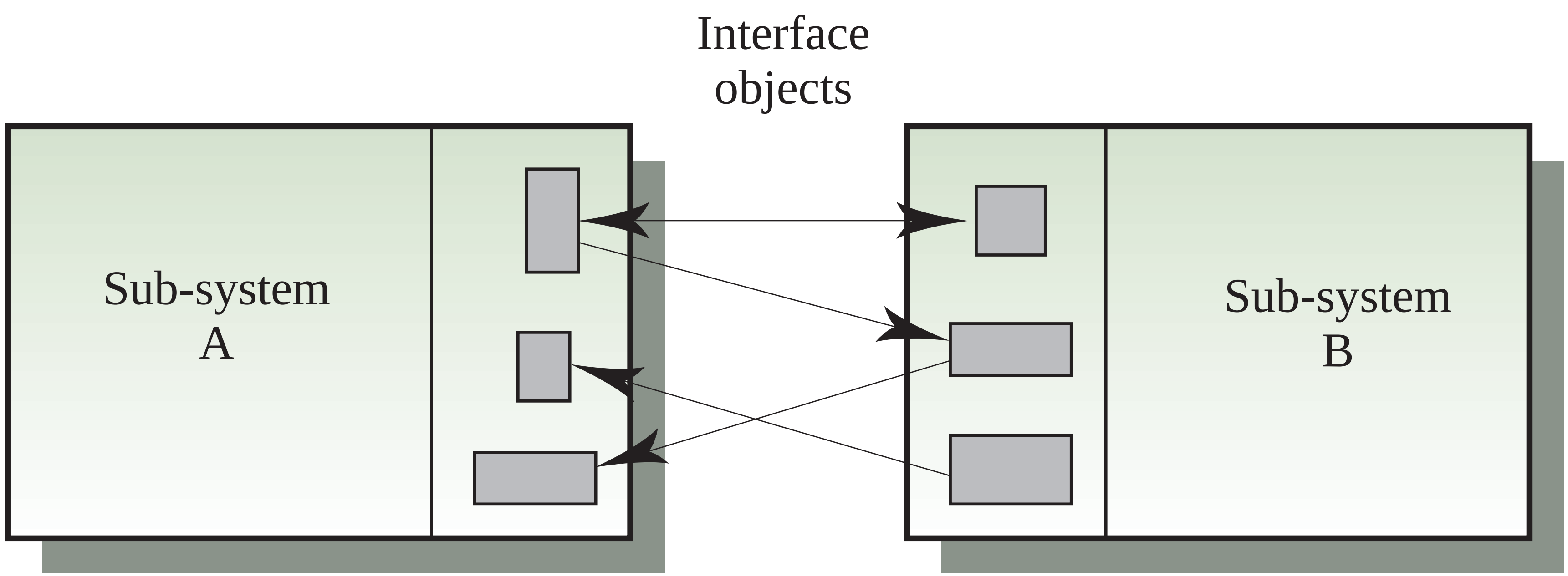
Topics covered

- Systematic algebraic specification
- Structured specification
- Error specification

Interface specification

- ⊗ Formal specification is particularly appropriate for defining sub-system interfaces. It provides an unambiguous interface description and allows for parallel sub-system development
- ⊗ Interfaces may be defined as a set of abstract data types or object classes
- ⊗ Algebraic specification is particularly appropriate for ADT specification as it focuses on operations and their relationships

Sub-system interfaces



From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Specification structure

- ⊗ Introduction
  - Introduces the sort (type) name and imported specifications
- ⊗ Informal description
  - Describes the type or object class operations
- ⊗ Signature
  - Defines the syntax of the type or class operations
- ⊗ Axioms
  - Defines axioms which characterize the behavior of the type

Specification format

< 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

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Specification of an array

- Replace with portrait slide

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 behavior, define the inspector 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

- ⊗ Replace with portrait slide

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]

Primitive constructors

- It is sometimes necessary to introduce additional constructors to simplify the specification
- The other constructors can then be defined using these more primitive constructors
- In the binary tree specification, a primitive constructor Build is added

Operations on a binary tree

Operation	**Description**
Create	Creates an empty tree.
Add (Binary_tree, Elem)	Adds a node to the binary tree using the usual ordering principles i.e. if it is less than the current node it is entered in the left subtree; if it is greater than or equal to the current node, it is entered in the right sub-tree.
Left (Binary_tree)	Returns the left sub-tree of the top of the tree.
Data (Binary_tree)	Returns the value of the data element at the top of the tree.
Right (Binary_tree)	Returns the right sub-tree of the top of the tree.
Is_empty (Binary_tree)	Returns true if the tree does not contain any elements.
Contains (Binary_tree, Elem)	Returns true if the tree contains the given element.

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Binary tree specification

⊗ Replace with portrait slide

Structured specification

- ⊗ Specifications should be constructed in a structured way. Other specifications should be reused whenever possible
- ⊗ Specification instantiation. A generic specification is instantiated with a given sort
- ⊗ Incremental specification. Use simple specifications in more complex specifications
- ⊗ Specification enrichment. A specification is constructed by inheritance from other specifications

Specification instantiation

```
CHAR_ARRAY: ARRAY

sort Char_array instantiates Array (Elem:=Char)
imports INTEGER
```

Incremental specification

- Develop a simple specification then use this in more complex specifications
- Try to establish a library of specification building blocks that may be reused
- In a graphical user interface, the specification of a Cartesian coordinate can be reused in the specification of a cursor
- Display operations are hard to specify algebraically. May be informally specified

Coord specification

COORD

sort Coord
imports INTEGER, BOOLEAN

Defines a sort representing a Cartesian coordinate. The operations defined on Coord are X and Y which evaluate the x and y attributes of an entity of this sort and Eq which compares two entities of sort Coord for equality.

Create (Integer, Integer) $\rightarrow$ Coord ;
X (Coord) $\rightarrow$ Integer ;
Y (Coord) $\rightarrow$ Integer ;
Eq (Coord, Coord) $\rightarrow$ Boolean ;

X (Create (x, y)) = x
Y (Create (x, y)) = y
Eq (Create (x1, y1), Create (x2, y2)) = ((x1 = x2) and (y1 = y2))

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Cursor specification

⊗ Replace with portrait slide

Specification enrichment

- ⊗ Starting with a reusable specification building block, new operations are added to create a more complex type
- ⊗ Enrichment can be continued to any number of levels. It is comparable to inheritance
- ⊗ Not the same as importing a specification
  - Importing makes a specification available for use
  - Enrichment creates a specification for a new sort
- ⊗ The names of the generic parameters of the base sort are inherited when a sort is enriched

Operations on New_list

Operation	Description
Create	Brings a list into existence.
Cons (New_list, Elem)	Adds an element to the end of the list.
Add (New_list, Elem)	Adds an element to the front of the list.
Head (New_list)	Returns the first element in the list.
Tail (New_list)	Returns the list with the first element removed.
Member (New_list, Elem)	Returns true if an element of the list matches Elem
Length (New_list)	Returns the number of elements in the list

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

New_list specification

NEW_LIST (Elem: [Undefined → Elem; .==. → Boolean])

sort New_List **enrich** List
imports INTEGER, BOOLEAN

Defines an extended form of list which inherits the operations and properties of the simpler specification of List and which adds new operations (Add and Member) to these.
See Figure 10.10 for a description of the list operations.

Add (New_List, Elem) → New_List
Member (New_List, Elem) → Boolean

Add (Create, v) = Cons (Create, v)
Member (Create, v) = FALSE
Member (Add (L, v), v1) = ((v == v1) **or** Member (L, v1))
Member (Cons (L, v), v1) = ((v == v1) **or** Member (L, v1))
Head (Add (L, v)) = v
Tail (Add (L, v)) = L
Length (Add (L, v)) = Length (L) + 1

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Multi-value operations

- Some operations affect more than one entity Logically, a function returns more than one value
- Stack pop operation returns both the value popped from the stack AND the modified stack
- May be modeled algebraically using multiple operations (TOP and RETRACT for a stack) but a more intuitive approach is to define operations which return a tuple rather than a single value

Queue operations

Operation	Description
Create	Brings a queue into existence.
Cons (Queue, Elem)	Adds an element to the end of the queue.
Head (Queue)	Returns the element at the front of the queue.
Tail (Queue)	Returns the queue minus its front element.
Length (Queue)	Returns the number of elements in the queue.
Get (Queue)	Returns a tuple composed of the element at the head of the queue and the queue with the front element removed

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Queue specification

QUEUE (Elem: [Undefined $\rightarrow$ Elem])

sort Queue **enrich** List
imports INTEGER

This specification defines a queue which is a first-in, first-out data structure. It can therefore be specified as a List where the insert operation adds a member to the end of the queue.
See Figure 10.12 for a description of queue operations.

Get (Queue) $\rightarrow$ (Elem, Queue)

Get (Create) = (Undefined, Create)
Get (Cons (Q, v)) = (Head (Q), Tail (Cons (Q, v)))

Error specification

- ⊗ Under normal conditions the result of an operation may be sort X but under exceptional conditions, an error should be indicated and the returned sort is different.
- ⊗ Problem may be tackled in three ways
  - Use a special distinguished constant operation (Undefined) which conforms to the type of the returned value. See array specification
  - Define operation evaluation to be a tuple, where an element indicates success of failure. See Queue specification
  - Include a special failure section in the specification

List with exception part

LIST (Elem)

sort List
imports INTEGER

See Figure 10.4

Create → List
Cons (List, Elem) → List
Tail (List) → List
Head (List) → Elem
Length (List) → Integer

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)
e**xceptions**
Length (L) = 0 fi f**ailure** (Head (L))

From: Sommerville, I. Software Engineering, 5th edition (Chap 10) Addison Wesley, 1995

Key points

- Algebraic specification is particularly appropriate for sub-system interface specification
- Algebraic specification involves specifying operations on an abstract data types or object in terms of their inter-relationships
- An algebraic specification has a signature part defining syntax and an axioms part defining semantics
- Formal specifications should have an associated informal description to make them more readable

Key points

- Algebraic specifications may be defined by defining the semantics of each inspection operation for each constructor operation
- Specification should be developed incrementally from simpler specification building blocks
- Errors can be specified either by defining distinguished error values, by defining a tuple where one part indicates success or failure or by including an error section in a specification