Formal 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 mathematical representation and analysis of software.
- Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.

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;
  - Formal methods are still hard to scale up to large systems.

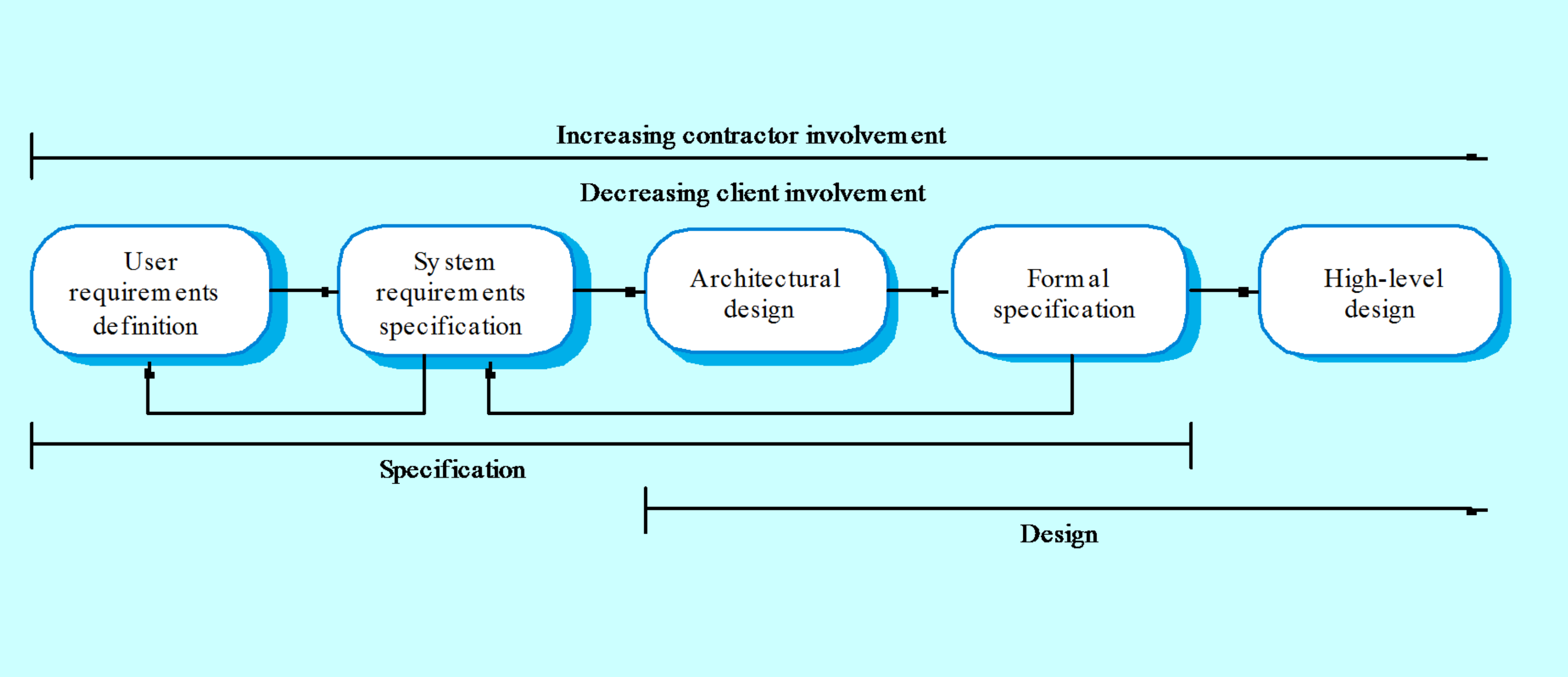
Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

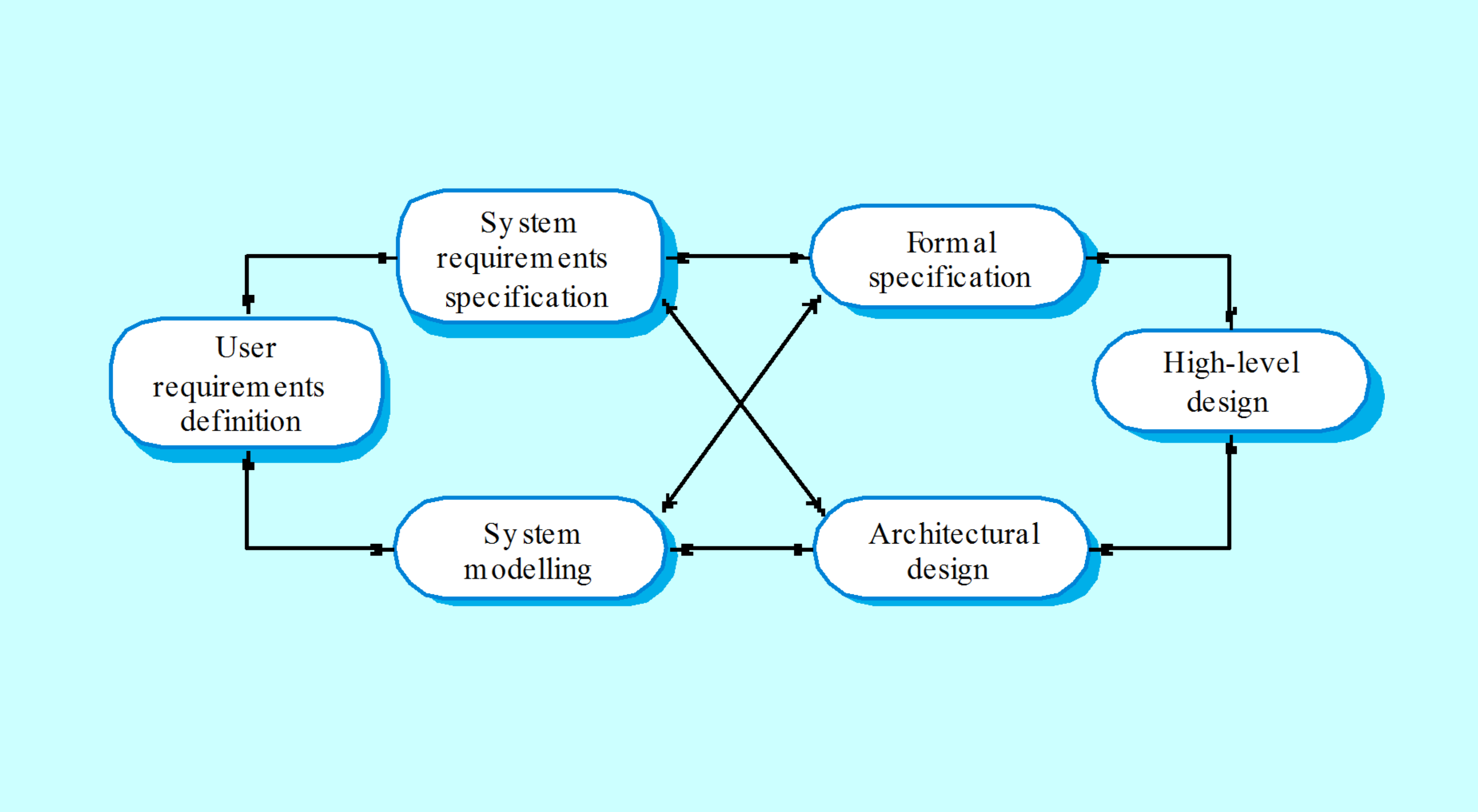
Specification in the software process

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

Specification and design


Increasing contractor involvement
Decreasing client involvement
User requirements definition
System requirements specification
Architectural design
Formal specification
High-level design
Specification
Design

Specification in the software process



User requirements definition

System requirements specification

System modelling

Formal specification

Architectural design

High-level design

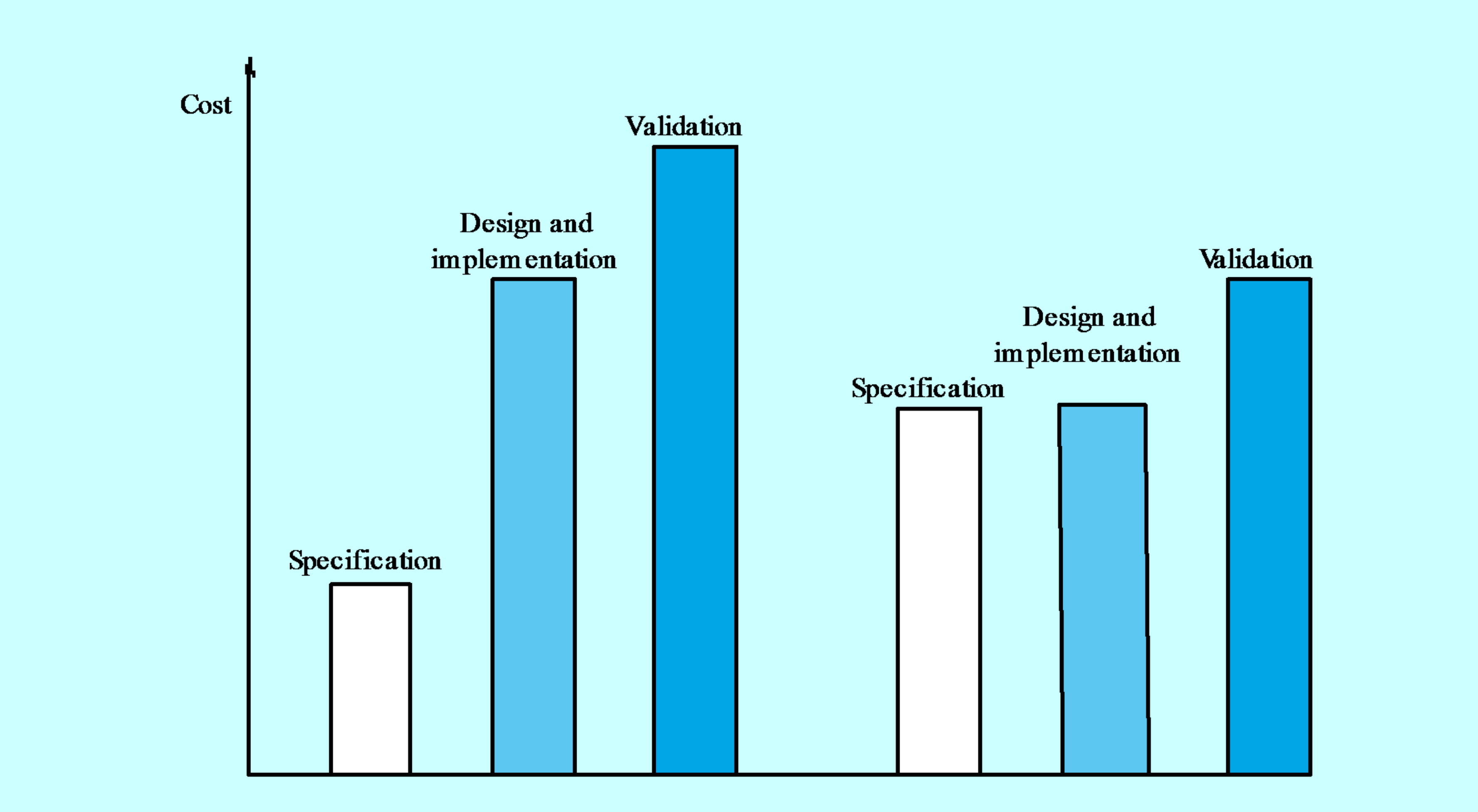
Use of formal specification

- 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 as made as the amount of rework due to requirements problems is reduced.

Cost profile

- The use of formal specification means that the cost profile of a project changes
  - There are greater up front costs as more time and effort are spent developing the specification;
  - However, implementation and validation costs should be reduced as the specification process reduces errors and ambiguities in the requirements.

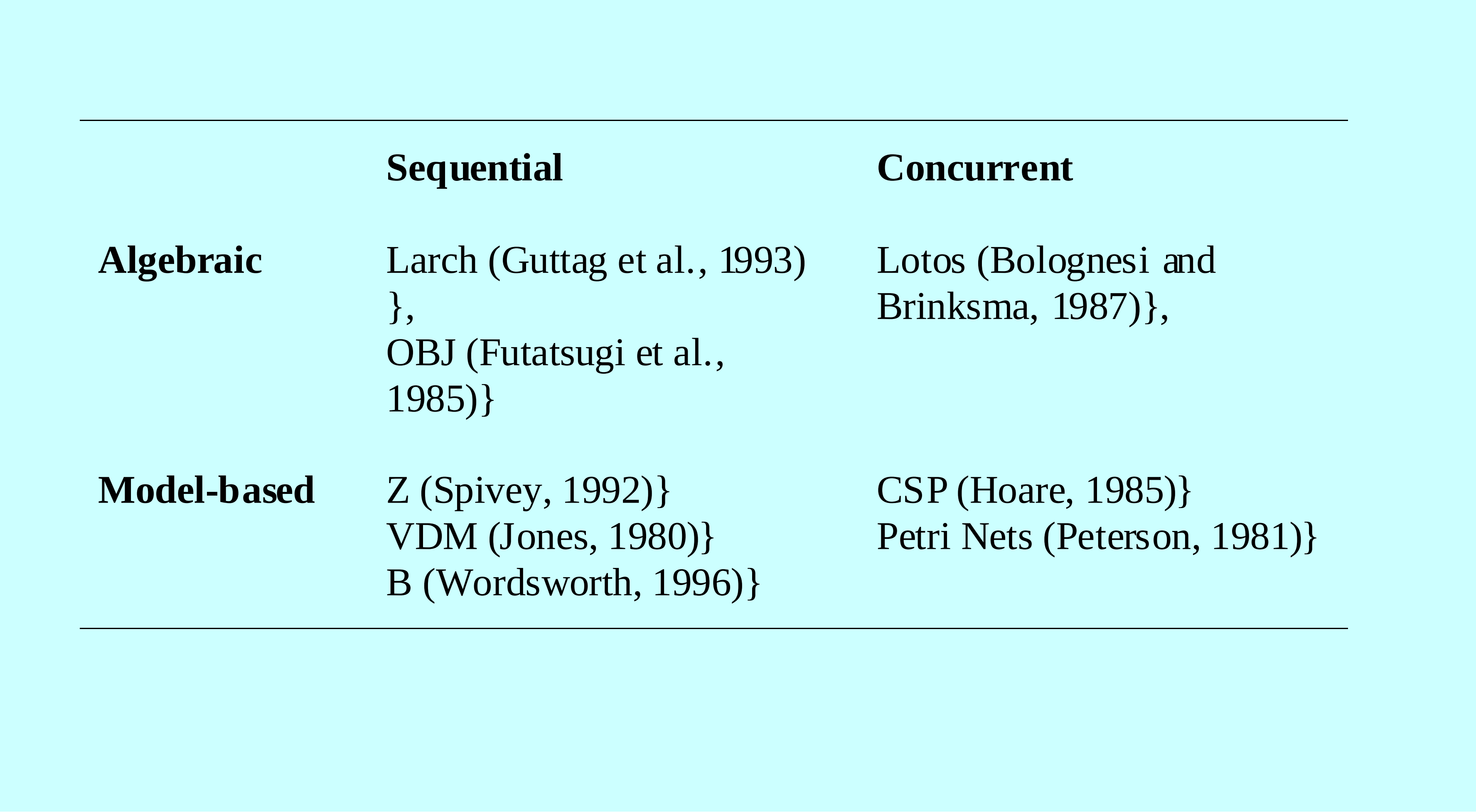
Development costs with formal specification



Specification techniques

- Algebraic specification
  - The system is specified in terms of its operations and their relationships.
- Model-based specification
  - 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



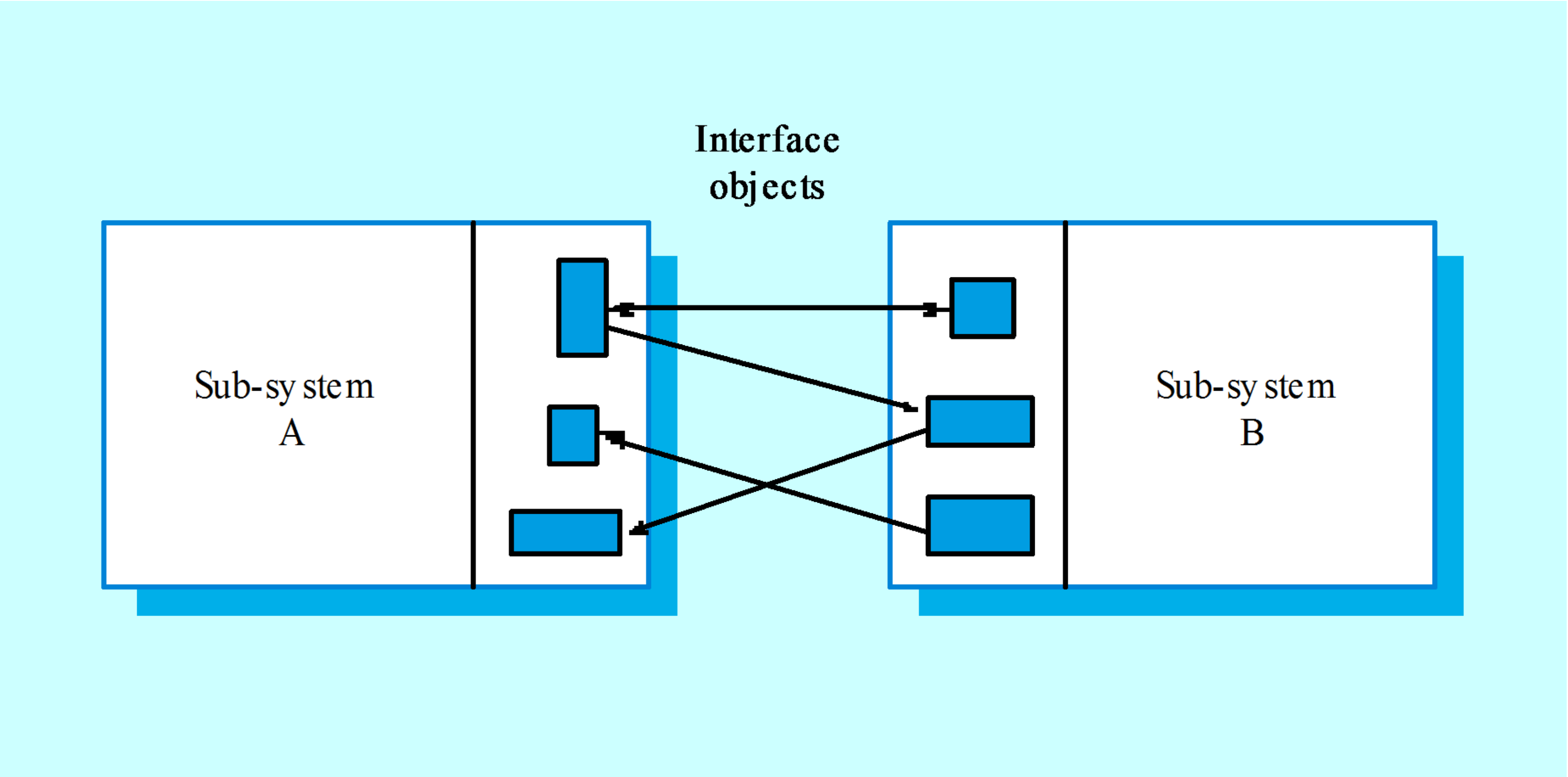
	Sequential	Concurrent
Algebraic	Larch (Guttag et al., 1993) }, OBJ (Futatsugi et al., 1985)}	Lotos (Bolognesi and Brinksma, 1987)},

Model-based	Z (Spivey, 1992)}
	VDM (Jones, 1980)}
	B (Wordsworth, 1996)}
		CSP (Hoare, 1985)}
		Petri Nets (Peterson, 1981)}

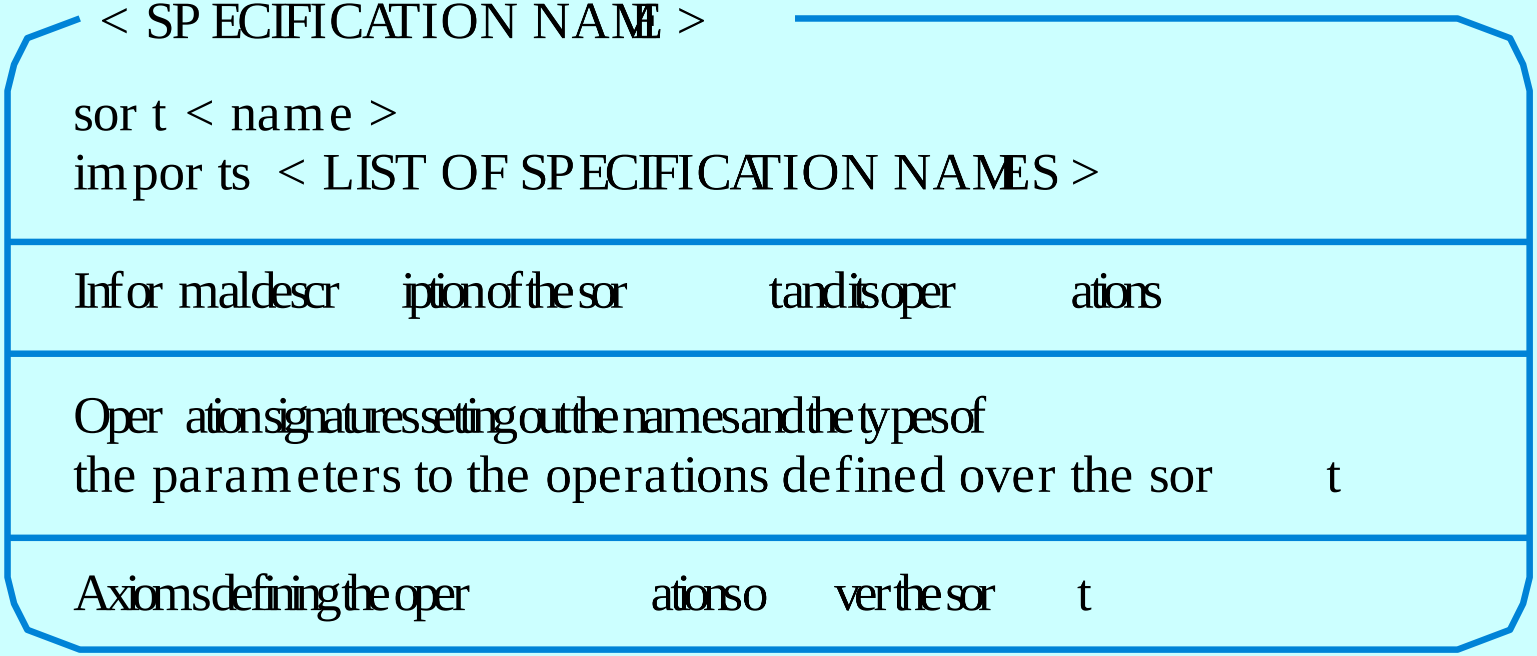
Interface specification

- 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 >

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.

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.

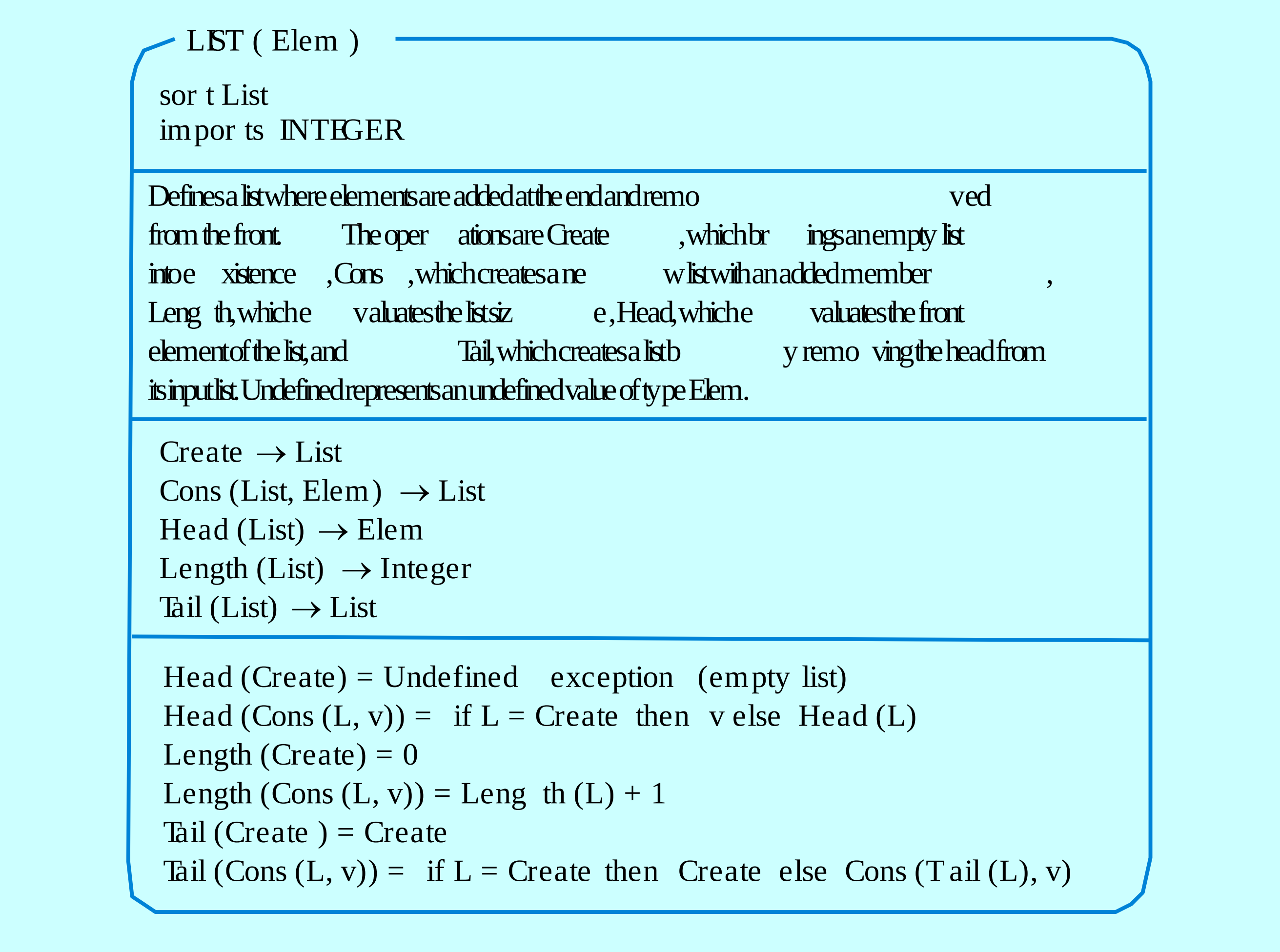
Specification operations

- 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.

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.

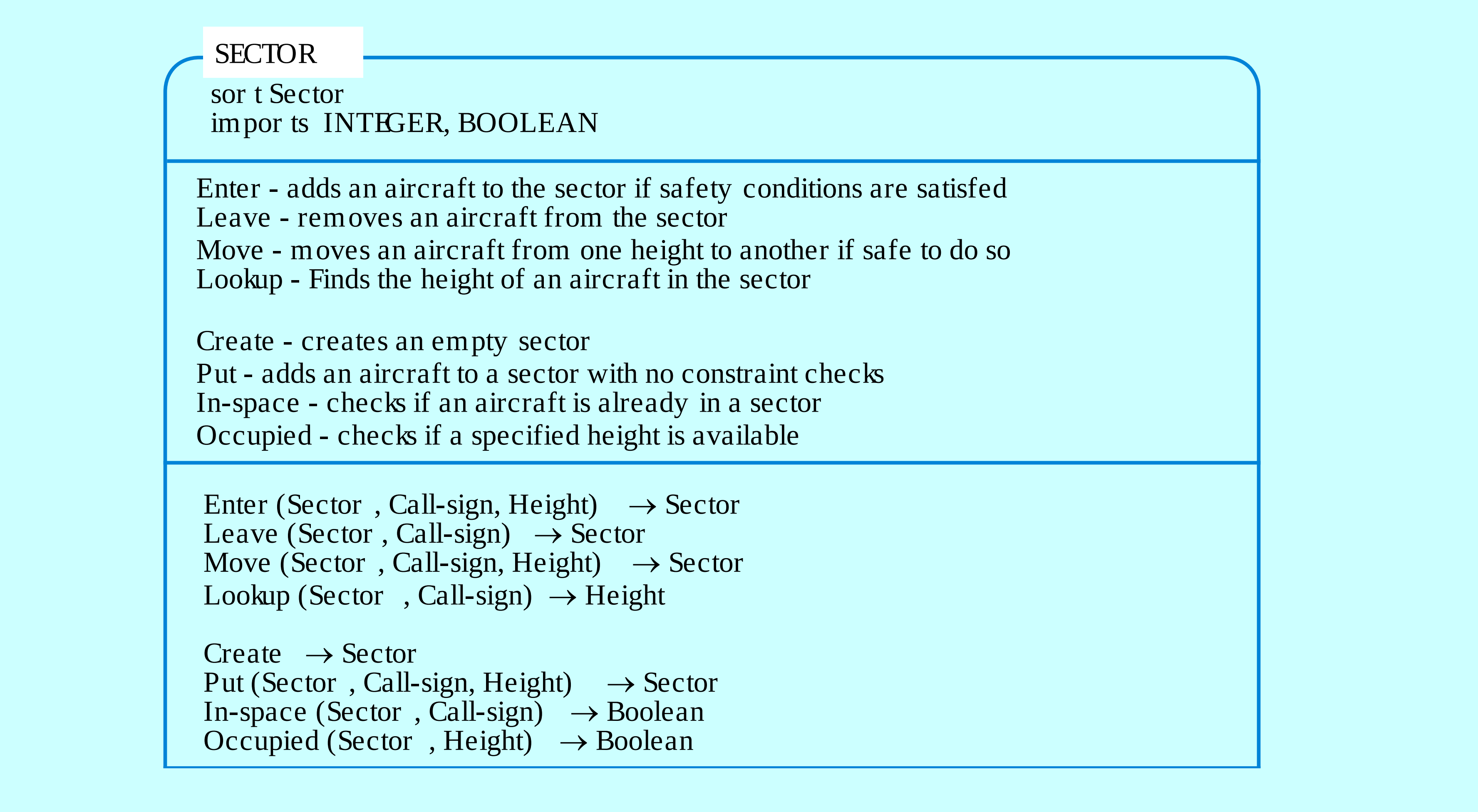
A sector object

- 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 (1)



SECTOR

sort Sector
imports INTEGER, BOOLEAN

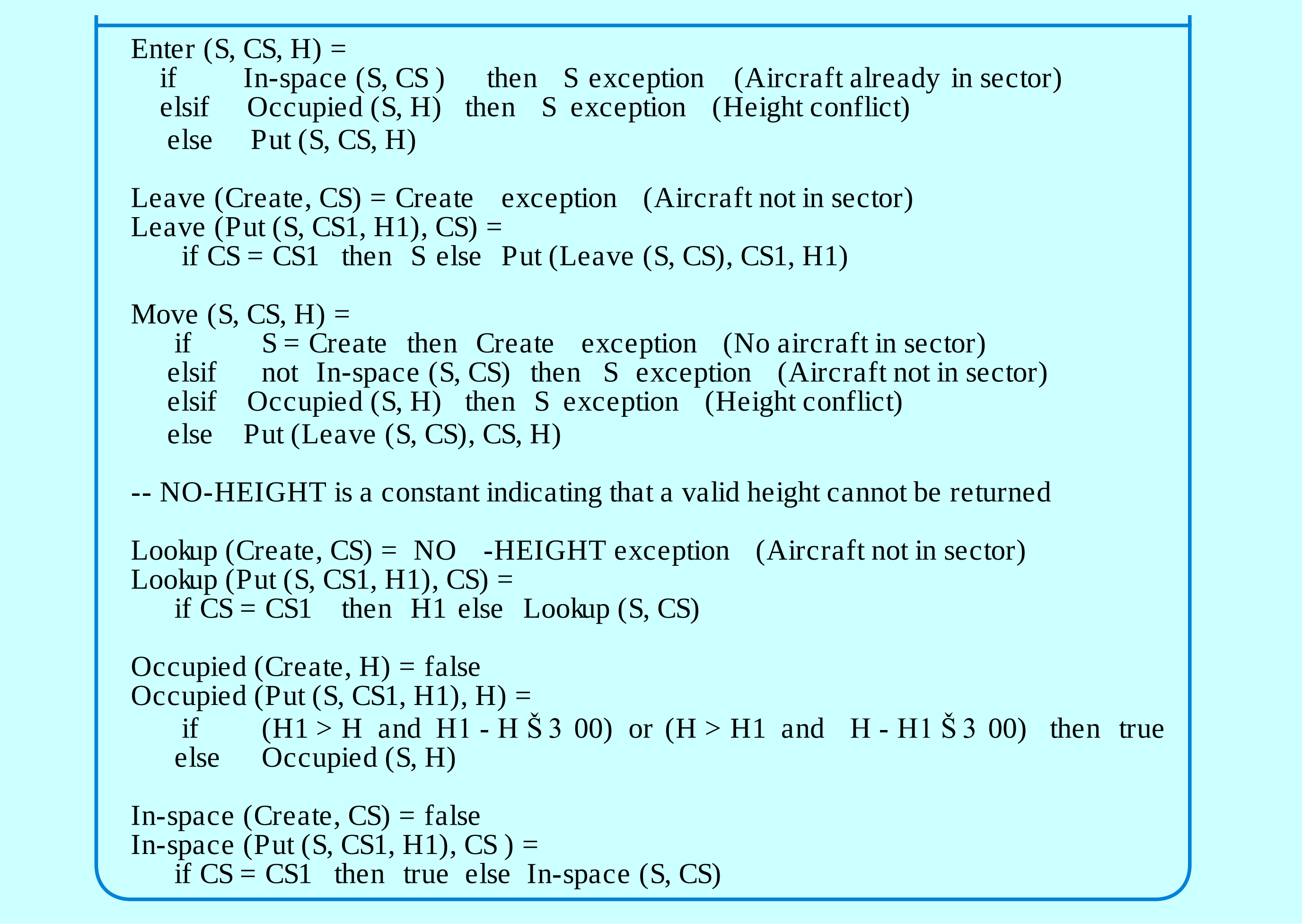
Enter - adds an aircraft to the sector if safety conditions are satisfied
Leave - removes an aircraft from the sector
Move - moves an aircraft from one height to another if safe to do so
Lookup - Finds the height of an aircraft in the sector

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

Sector specification (2)



```
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 3  00)  or  (H > H1  and    H - H1 Š 3  00)   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)
```

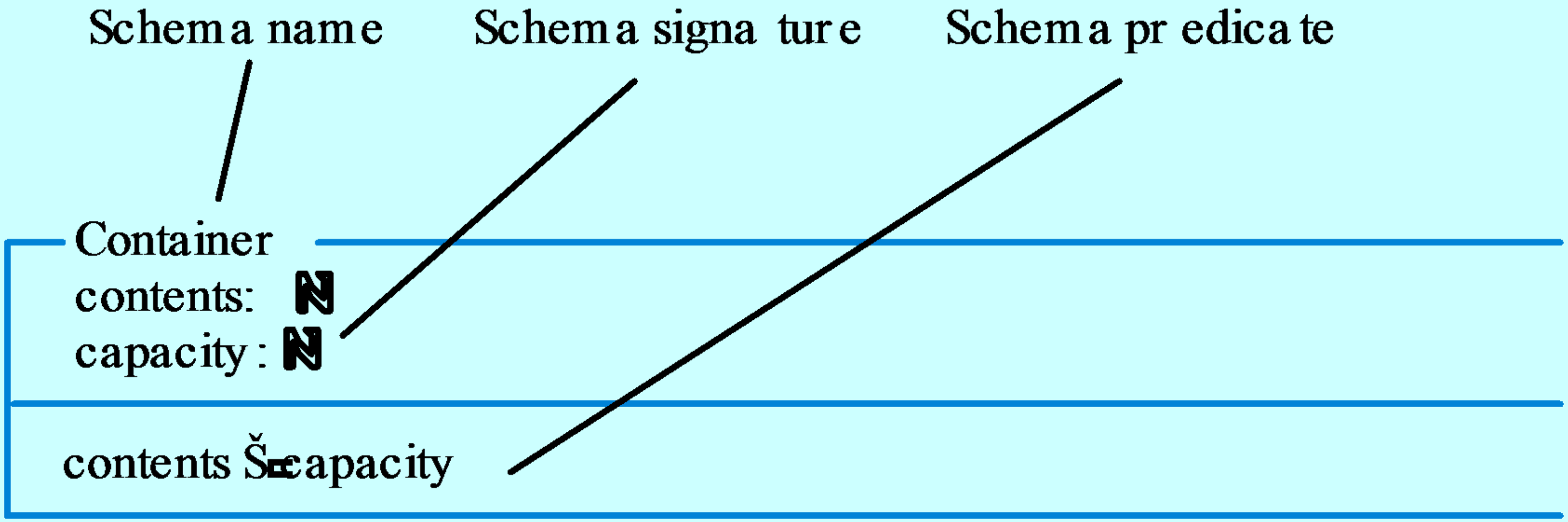
Specification commentary

- 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.

The structure of a Z schema



Schema name

Schema signature

Schema predicate

Container

contents: ℕ

capacity : ℕ

contents Šcapacity

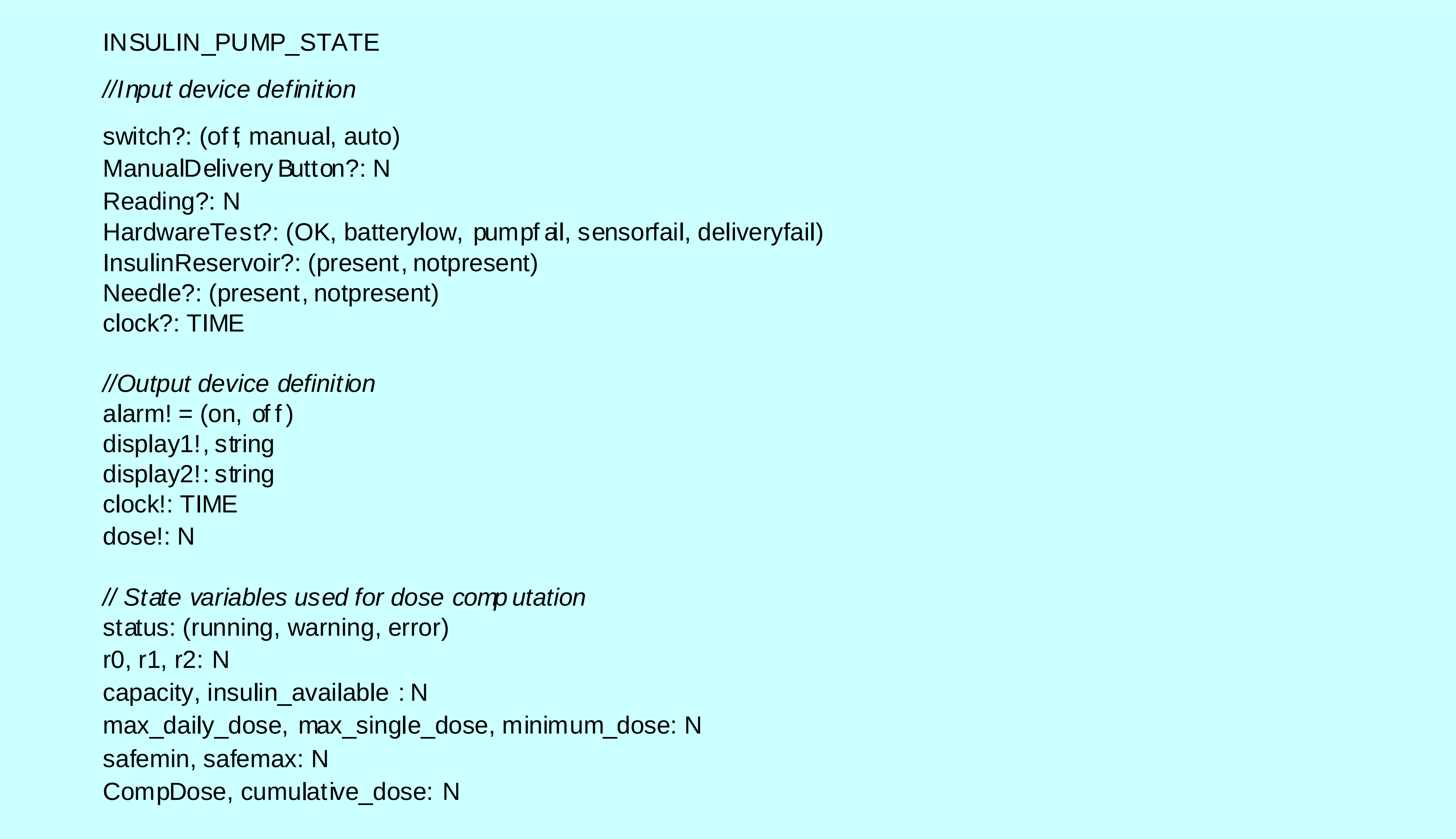
Modelling the insulin pump

- The Z schema for the insulin pump declares a number of state variables including:
  - Input variables such as switch? (the device switch), InsulinReservoir? (the current quantity of insulin in the reservoir) and Reading? (the reading from the sensor);
  - Output variables such as alarm! (a system alarm), display1!, display2! (the displays on the pump) and dose! (the dose of insulin to be delivered).

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 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 schema



```
INSULIN_PUMP_STATE

//Input device definition

switch?: (off, manual, auto)
ManualDeliveryButton?: N
Reading?: N
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!: N

// 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
```

State invariants

r2 = Reading?

dose! Š insulin_available

insulin_available Š 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 (1)

RUN

ΔINSULIN_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

∨

RUN schema (2)

// 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 Š 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 Š max_single_dose * 4 ⇒  status’ = warning ∧
display1! = “Insulin low”

r1’ = r2
r0’ = r1
```

Sugar OK schema

```
SUGAR_OK
        r2 •  safemin ∧ r2 Š safemax

        // sugar level stable or falling

        r2 Š r1 ⇒ CompDose = 0

        ∨

        // sugar level increasing but rate of increase falling

        r2 > r1 ∧ (r2-r1) < (r1-r0) ⇒ CompDose = 0

        ∨

        // sugar level increasing and rate of increase increasing compute dose
        // a minimum dose must be delivered if rounded to zero

        r2 > r1 ∧ (r2-r1) □(r1-r0) ∧ (round ((r2-r1)/4) = 0)  ⇒
                        CompDose = minimum_dose
        ∨
        r2 > r1 ∧ (r2-r1) •  (r1-r0) ∧ (round ((r2-r1)/4) > 0) ⇒
                        CompDose = round ((r2-r1)/4)
```

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.

Key points

- Algebraic techniques are suited to interface specification where the interface is defined as a set of object classes.
- Model-based techniques model the system using sets and functions. This simplifies some types of behavioural specification.
- Operations are defined in a model-based spec. by defining pre and post conditions on the system state.