
Formal Specifications

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 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 hard to scale up to large systems

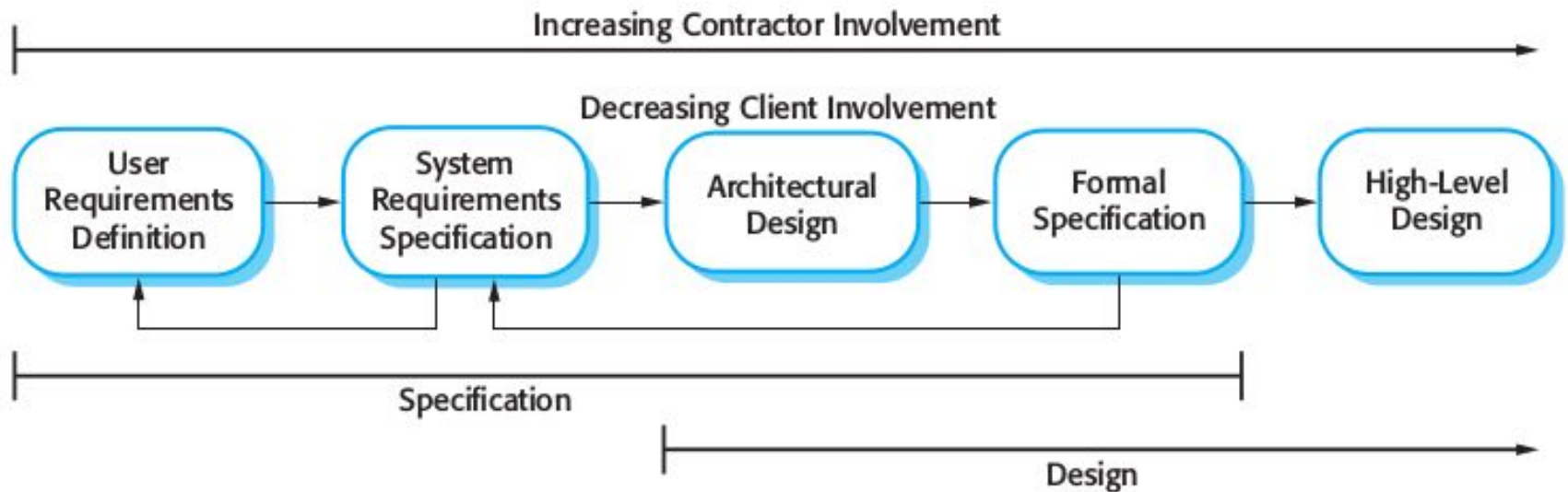
Use of formal methods

- Formal methods have limited practical applicability
- Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems
- In this area, the use of formal methods 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 mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and 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

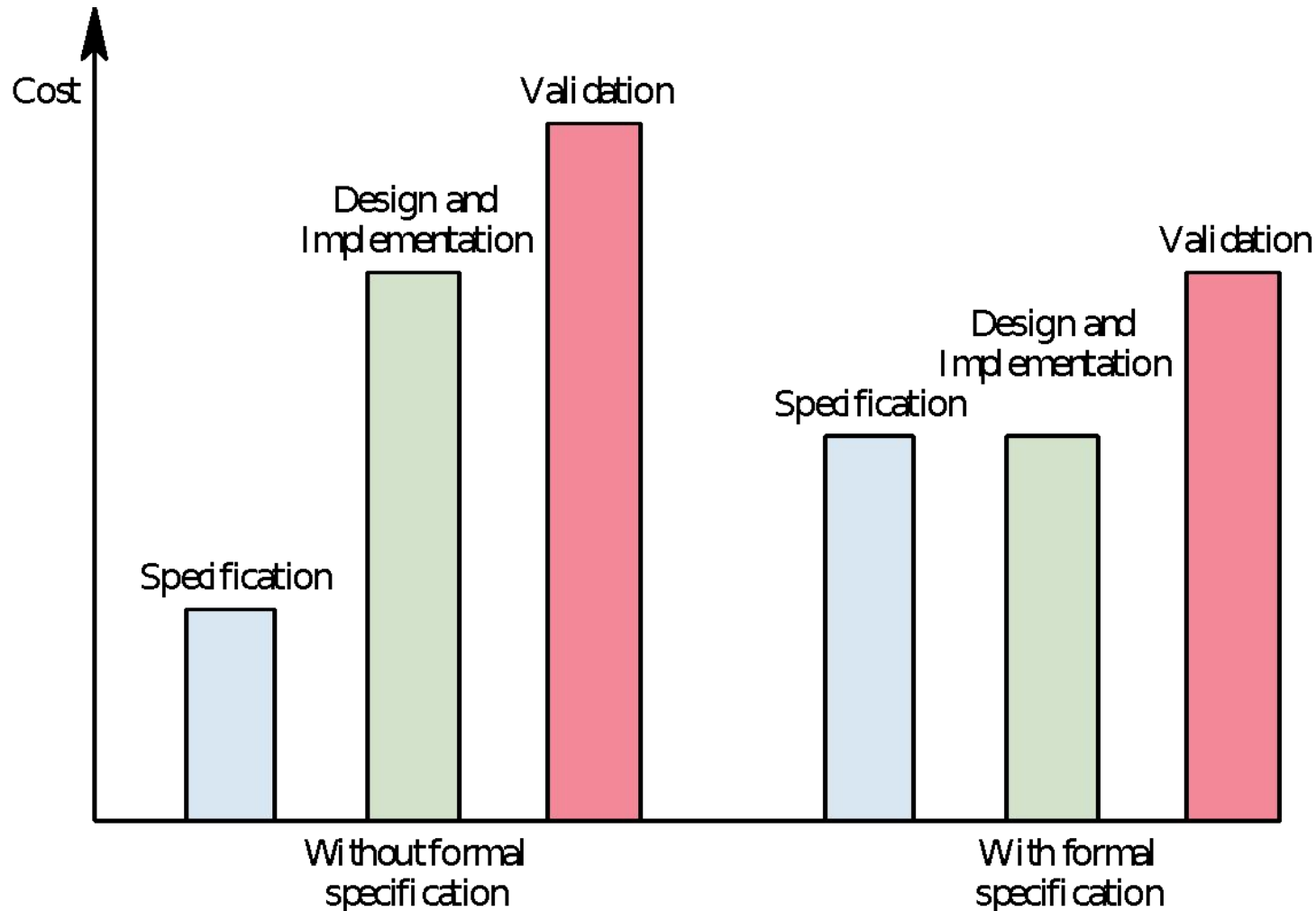
Formal specification languages

	Sequential	Concurrent
Algebraic	Larch (Guttag, Horning et al., 1985; Guttag, Horning et al., 1993), OBJ (Futatsugi, Goguen 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)

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

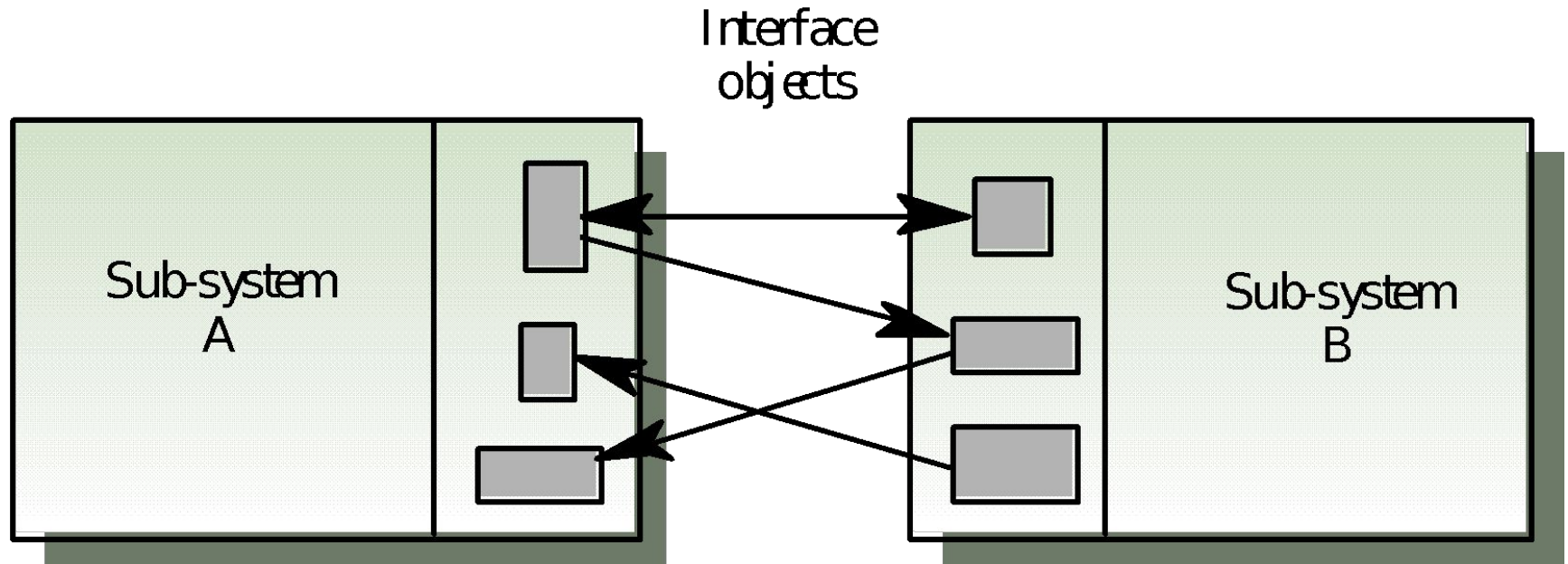
Development costs with formal specification



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



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

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 \rightarrow List
Cons (List, Elem) \rightarrow List
Head (List) \rightarrow Elem
Length (List) \rightarrow Integer
Tail (List) \rightarrow 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

- 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

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)  
  elif 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)  
  elif not In-space (S, CS) then S exception (Aircraft not in sector)  
  elif 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 ≤ 300) or (H > H1 and H - H1 ≤ 300) then true  
  else Occupied (S, H)
```

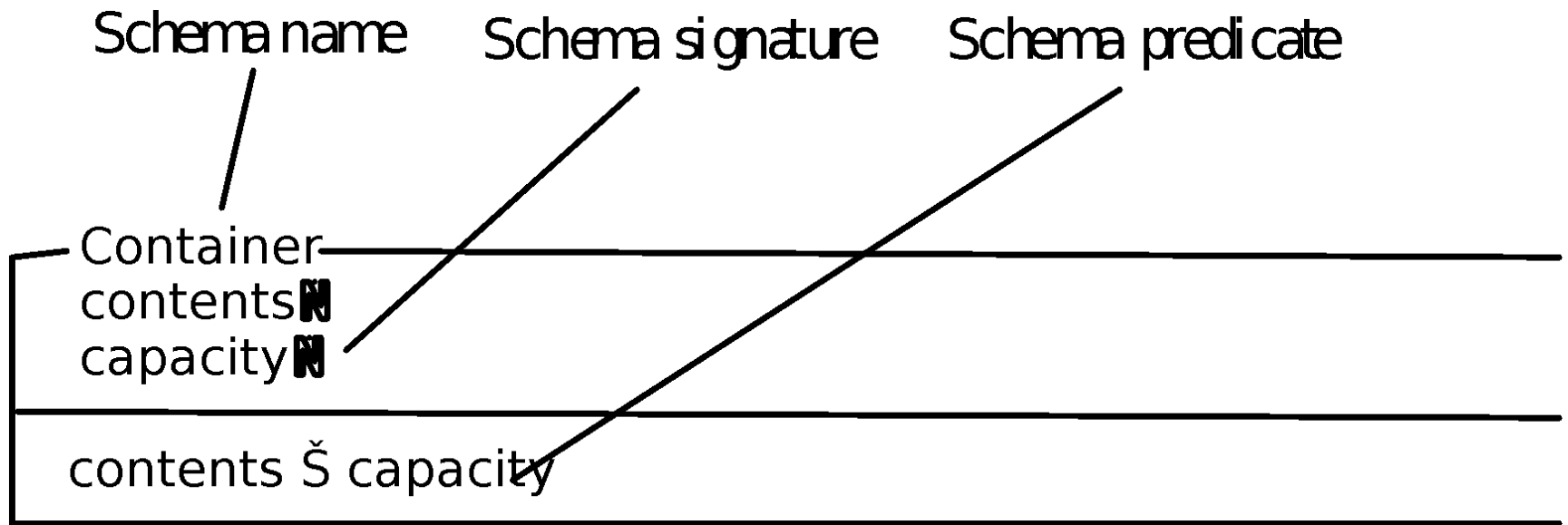
In-space (Create, CS) = false

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

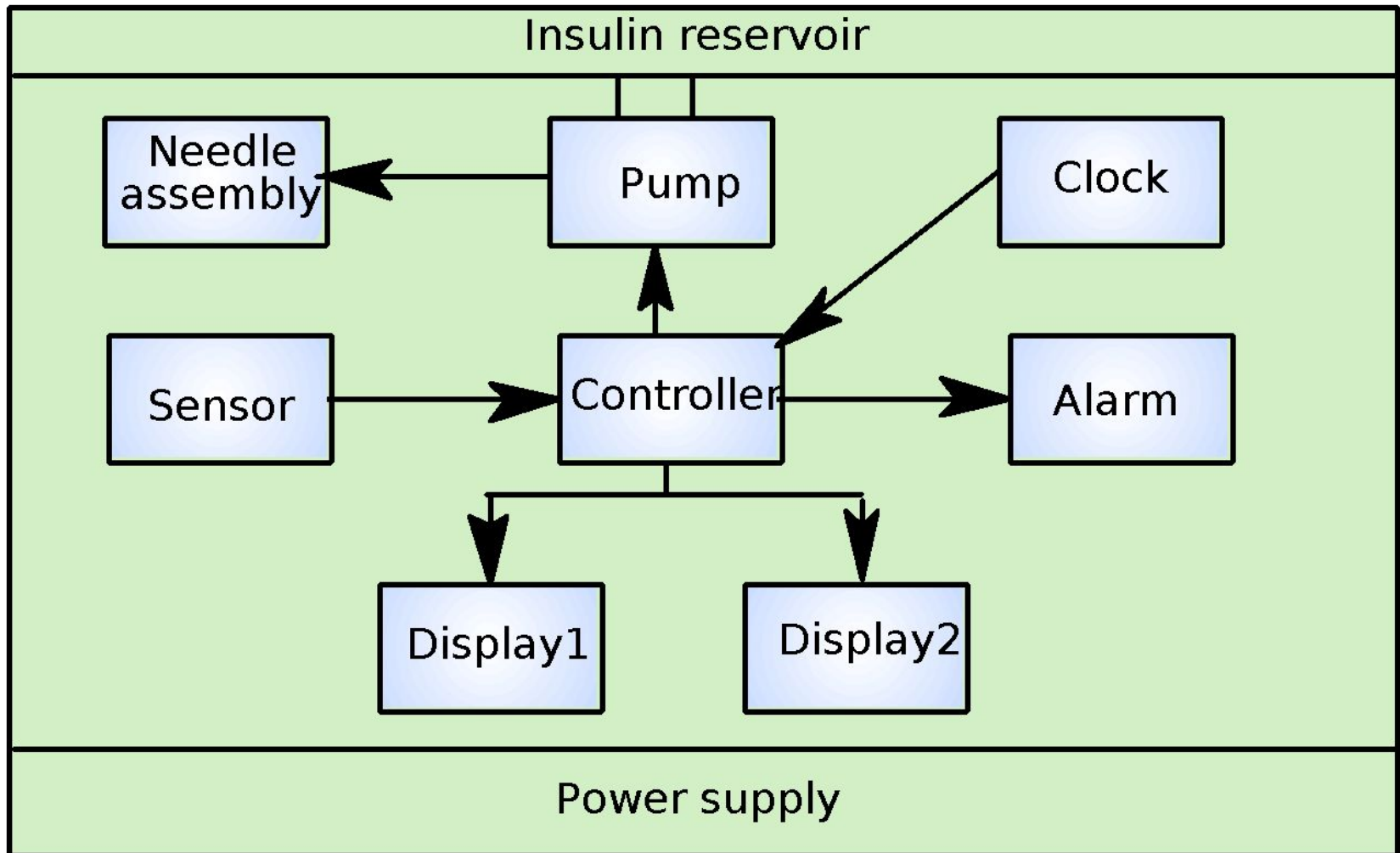
Behavioural specification

- 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



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, names followed by a ! are 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 (see Chapters 16 and 17)
 - display1! shows the status of the insulin reservoir.

Insulin pump schema

Insulin_pump

reading? : \mathbb{N}

dose, cumulative_dose: \mathbb{N}

r0, r1, r2: \mathbb{N} // used to record the last 3 readings taken

capacity: \mathbb{N}

alarm!: {off, on}

pump!: \mathbb{N}

display1!, display2!: STRING

dose \checkmark capacity \wedge dose \checkmark 5 \wedge cumulative_dose \checkmark 50

capacity \square 40 \Rightarrow display1! = ""

capacity \checkmark 39 \wedge capacity \square 10 \Rightarrow display1! = "Insulin low"

capacity \checkmark 9 \Rightarrow alarm! = on \wedge display1! = "Insulin very low"

r2 = reading?

DOSAGE schema

DOSAGE

Δ Insulin_Pump

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

Output schemas

DISPLAY
 Δ Insulin_Pump

$\text{display2!} = \text{Nat_to_string}(\text{dose}) \wedge$
 $(\text{reading?} < 3 \Rightarrow \text{display1!} = \text{"Sugar low"}) \vee$
 $\text{reading?} > 30 \Rightarrow \text{display1!} = \text{"Sugar high"} \vee$
 $\text{reading?} \square 3 \text{ and } \text{reading?} \checkmark 30 \Rightarrow \text{display1!} = \text{"OK"})$

ALARM
 Δ Insulin_Pump

$(\text{reading?} < 3 \vee \text{reading?} > 30) \Rightarrow \text{alarm!} = \text{on} \vee$
 $(\text{reading?} \square 3 \wedge \text{reading?} \checkmark 30) \Rightarrow \text{alarm!} = \text{off}$

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

Key points

- Formal specification techniques are most applicable in the development of critical systems and standards.
- Algebraic techniques are suited to interface specification and model-based techniques are suited for behavioural specification.