CHAPTER 7 – Formal Specification

Introduction
- When, Why and What?
  + Classification
  + Unambiguous, Consistent, Complete
    - Design by contract + Testing

Input/Output Specifications
- Pre- and postconditions + invariants

Algebraic Specifications
- Rewrite rules

Logic-Based (a.k.a. Model-Based) Specifications
- Data-model constraints + State transitions

State-Based Specifications
- Statecharts

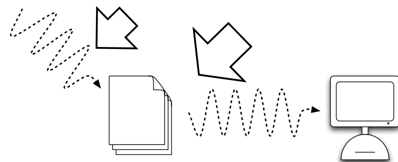
Conclusion
- Cleanroom Development
- Correctness & Traceability

Literature

- [Ghez02a] In particular, chapters "Specification" and "Verification - Analysis"
- [Somm04a] In particular, chapters "Formal Specification" & "Verification and Validation"
- [Pres01a] In particular chapters "Formal Methods" & "Cleanroom Software Engineering"
- [Stev00a] P. Stevens, R. Pooley, Using UML - Software Engineering with Objects and Components, Addison-Wesley 2000. Chapters 11 & 12 on State Diagrams.
- [Bind00a] Testing Object-Oriented Systems — Models, Patterns, and Tools, Addison-Wesley, 2000. Chapter 7 & 8 on State Machines and UML

Web Resources
- Formal Methods Europe
  ➡ http://www.fmeurope.org/
- Department of Defence - Data & Analysis Centre for Software
  ➡ https://www.dacs.dtic.mil/databases/url/key.php?keycode=53
- The World-Wide Web Virtual Library from Oxford University
  ➡ http://vl.fmnet.info/

When Formal Specification?



Formal Specification is used for
- (detailed) design
  specify key properties of final system
- analysis
  specify "what", not "how"

Why Formal Specification?

Software projects rely more and more on "Buy" than on "Build"
- Cheaper, more reliable, ...
- Companies focus in-house development on core business
  + Buy 3rd party components for functionality outside the core
- 3rd party components evolve
  + require well-specified interface

Buy vs. Build
- But if we buy we need to specify
  + clearly
  + unambiguously
  + completely

⇒ **Formally**

Formal Specification Trade-Off's

Formal specs cost ...	however ...
• high cost of specification	• but better verifiability + safety, reliability, security, ... • lower cost of implementation
• require highly trained staff + (background in mathematics, logics)	• only simple mathematical skills and elementary logic
• customers cannot understand specification	• depends on the customers/ the specification technique &style • prototypes can be generated/ formal specs can be animated
• not applicable for all parts (GUI, ...)	• are these parts critical?
• lack of tools	• more and more tools become available
• formal specs don't scale well	• same for programming languages • use formal specs only for the critical parts

Why do we Care?

It is possible to build high-quality products without formal specifications!
(And it is possible to build low-quality products with formal specifications)

Fact
- For most systems it is more cost effective to apply other techniques (reviews, tests, ...)
  + business systems, information systems, ...
  + Most software development is in that area

Fact
- For some areas, the benefits more than outweigh the costs
  + high-risk systems: human lives depend on the software (because reliability is such a big issue)
  + embedded systems: software controlling hardware (because components evolve at different rates)
  + standards: defining information exchange protocols (because the same specification is reused by a lot of implementations)

Presence of standard implies that lots of people "buy instead of build"
Sooner or later you will be confronted with one of these

What are Formal Specifications?

What is a ...

- **Specification**. A description of desired system properties.
  + Preferably the "what" and not the "how"

- **Informal specification**. Specification in natural language.
  + augmented with figures, tables, examples, scenarios

- **Semi-formal specification**. Specification based on a notation with precise syntax but loose semantics.
  + e.g. UML class & sequence diagrams

- **Formal Specification**. Specification based on a formal model with precise syntax & semantics.

Consistent / Complete / Unambiguous

Formal model will make a formal specification ... [1]

Consistent
- no contradictions in the specification

Completeness
- all properties are defined in terms of known concepts

Unambiguous
- misinterpretations are impossible
- (sometimes also) minimal: there is only one way to express a certain property

[1] Only for the specified properties

Testing and Design by Contract

Formal foundation: formal syntax + formal semantics
- Possible to prove that a given system satisfies the specification

A system is correct with respect to its specification !
Note: errors and omissions in the specification are still possible

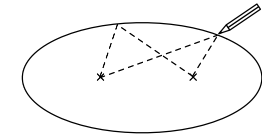
Testing
- Formal specifications ⇒ black-box testing
  test-cases: complete coverage, thus highest probability of finding mistakes

Design by Contract
- Formal specifications ⇒ natural pre- and post conditions

Declarative vs. Operational

(see [Ghez02a] "Classification of Specification Styles")

Operational
- = describe the desired behavior
- Select two points P1 and P2 on a plane
- Select a string of a certain length and fix the ends to P1 and P2 respectively
- Position a pencil against the string
- Move the pen clockwise, keeping the string tightly stretched, until you reached the point where you started drawing



Declarative (a.k.a. descriptive)
- = describe the desired properties
- $\{(x, y) \in (Real \times Real) \mid ax^2 + by^2 + c = 0; a, b, c \in Real\}$

Classification (i)

A) Input/Output Specifications
- include logic assertions (pre & post-conditions + invariants) inside an algorithm
  + via formal reasoning one can "prove" these assertions
  + declarative, but with the aim of program verification
- Examples: theorem proving to verify that libraries work as intended

B) Algebraic Specifications
- system is described in terms of operations one can apply on types
  + via rewrite rules one can deduce the result of expressions
  + operational in nature
- Examples: Abstract Data Types, Larch, OBJ, relational algebra

Historical interest: show the roots of formal specifications
- You should understand why it is useful

Classification (ii)

C) Logic-based Specifications (a.k.a. Model-based)
- system is described using mathematical entities (e.g. sets)
  + via logic one declares propositions that must hold
  + declarative in nature
- Examples: Z, B, VDM, UML's constraint language OCL

D) State-based Specifications
- behaviour of system is described in terms of a state-machine
  + depending on external events system switches internal state
  + operational in nature
- Examples: Petri-nets, StateCharts (part of UML)

Practical interest: such specifications are widely used today
- You should be able to read (and write) such specifications

A) Input/Output Specifications

- include logic assertions (pre & post-conditions + invariants) inside an algorithm
- verify termination and correctness by stepwise formal reasoning

Example: Input/Output Specification for a binary search procedure

```
procedure Binary_search (Key : ELEM ; T: ELEM_ARRAY;
    Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
    T'LAST - T'FIRST >= 0 and   -- not empty
        for_all i, -- universal qualifier
            T'FIRST <= i <= T'LAST-1, T (i) <= T (i + 1) --sorted

Post-condition
    ( Found and T (L) = Key)  or
        ( not Found and
        not (exists i, -- existential qualifier
            T'FIRST <= i <= T'LAST, T (i) = Key ))
```


Intermediate Assertions

```
01.    Bott := T'FIRST; Top := T'LAST ;
02.    L := ( T'FIRST + T'LAST ) mod 2; Found := T( L ) = Key;
03.    -- 1 . ASSERT (Found and T(L) = Key) or ((not Found)
04.    -- and (not Key in T(T'FIRST..Bott-1, Top+1..T'LAST)));
05.    while Bott <= Top and not Found loop
06.         Mid := (Top + Bott) / 2;
07.         if T( Mid ) = Key then
08.            Found := true; L := Mid;
09.         --  2. ASSERT Key = T(Mid) and Found;
10.         elsif T( Mid ) < Key then
11.            --  3. ASSERT not Key in T(T'FIRST..Mid);
12.            Bott := Mid + 1;
13.         --  4. ASSERT not Key in T(T'FIRST..Bott-1);
14.         else
15.            --  5. ASSERT not Key in T( Mid..T'LAST );
16.            Top := Mid - 1;
17.         --  6. ASSERT not Key in T(Top+1..T'LAST);
18.    end if;
19.    end loop;
```


Verification

Goal:

- prove that post-condition is always satisfied when pre-condition is true

Termination?

- While loop terminates if Found or Bott > Top
- If an element = key exists, Found is set true
- In a loop execution either Found := true, Bott >> or Top <<
- Initially, Top > Bott thus (if Found remains false) eventually Bott > Top

Correctness?

- Loop invariant is true on entry to the loop.
- Assertion 2 follows because of the successful test Key = Mid
- Assertion 3 follows because the array is ordered. If T (Mid) < Key all values up to T (Mid) must also be less than the key
- Assertion 4 follows by substituting Bott-1 for Mid
- Assertions 5 and 6. Similar argument to 3 and 4
- After loop execution, either the key has been found or there is no value in the array which has been searched which matches the key. However, Bott > Top so all the array has been searched

⇒Therefore, the binary search routine code conforms to its specification

Consistent / Complete / Unambiguous

When is an input/output specification ...

- Consistent ?
  ±= termination
- Complete ?
  ±= correctness
- Unambiguous ?
  + Are all parts of pre- and postcondition necessary
  + Less relevant in practice, it shows that an interface is minimal !

Input/Output Specifications Revisited

Input/Output Specifications

- Are particularly suitable for program verification: crucial parts and/or libraries
  + Verify termination and correctness
- Express input and output as assertions
  + Pre- & postconditions
    ➡ Design by Contract
    ➡ (Black-box) Testing: Condition testing

B) Algebraic Specifications

- Specify properties of data type via relationships between the applicable functions
- Afterwards one can derive or verify other properties
- Goal: show interface is well designed + optimization properties

```
algebra StackSpecification
   Sorts Stack, Integer, Boolean
   Operations

      push (Stack, Integer) :- Stack        ⟼ Command
      pop (Stack) :- Stack                    function

      top (Stack) :- Integer               ⟼ Query function
      isEmpty (Stack) :- Boolean

      new () :- Stack         ⟼ Creator function
   Axioms
   For any x: Integer, s: Stack
      top(push(s, x)) = x
      pop(push(s, x)) = s
      empty(new()) = true
      empty(push(s, x)) = false
```


Verifications of Properties

Example of Usage

What is the result of an expression ?

```
(top (pop (push (push (new (), 3), 7))))
---
(top ...........(push (new (), 3) .....) -- pop(push(s, x)) = s
 ..............................3....... -- top(push(s, x)) = x
```

Can you switch the order of Pop and Push ?
(Switching the order of operations for optimization purposes)

```
(top(pop(push(push(new(),3),7))))
    =? (top(push(pop(push(new(),3)),7)))
---
(top(push(new(),3))) =? (top(push(new(),7)))
3 =? 7
false
```

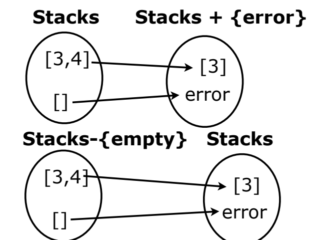

Functions in Algebraic Specifications

Operations

- The “Operations” section lists mathematical functions!
  + no side-effects, no state-changes
    - example: push (s, x) “copies” s before pushing the element x

Errors

- How to deal with error
  + (e.g., pop on an empty stack)
- a special “Error” value is always included in the range of a function
  + each newly defined function must include this special case
- include a section Preconditions which constrain the domain of the function
  + it is impossible to apply the function



Towards Design by Contract

- Design by Contract is "practical derivation", mixing Algebraic Specifications and Input/Output Assertions

Algebraic Specification
- To assure completeness and consistency of class interface
  + In mathematical terms!

Pre-conditions
- To constrain the domain of each operation in the interface
  + Thus to specify the input condition

Post-conditions
- To deal with computer functions, i.e. function with side-effects
  + An axiom in an algebraic specification becomes a post-condition

Works well with almost homogenous interfaces
(Sorts include one type to be specified and the rest are primitive types)

Consistent / Complete / Unambiguous

Consistent
- It is never possible to deduce contradictions (i.e. "true = false"?)
  + For instance, if we add an axiom
    empty(push(push(s, x), y)) = true
    it is possible to deduce "true = false"

Complete
- Can all query expressions be reduced?
  (query expression = outermost function is a query function)
  + For instance, if we remove an axiom
    pop(push(s, x)) = s
    there are query expressions which cannot be reduced

Unambiguous
- Are all functions and axioms required ?
  Less relevant in practice, it shows that an interface is minimal !

In general, these questions are *undecidable*.
For a particular algebra, these properties must be proven manually!

Algebraic Specifications Revisited

Algebraic Specifications
- Are particularly suitable for specifying (almost homogeneous) interfaces
  + Verify
    - consistency : false = true cannot be inferred
    - completeness : all expressions can be reduced
    - ambiguity (optional): interface is minimal
  + This interface is "proven" to be a good one
- Foundation for Abstract Data Types, thus Objects
- However, operations on objects have side-effects
  + Constraints on input-domain (pre-condition)
    & axioms (post-condition)
    ➡ Design by Contract
    ➡ (Black-box) Testing: Condition testing

C) Logic-Based Specifications

As an example, we choose Z-syntax
- specify data-structures via typed sets
- specify constraints via 2-valued (true/false) logic propositions

```
┌ Container ──────────┐
│ contents: Integer
│ capacity: Integer
├─────────────────────
│ contents ≤ capacity
└─────────────────────┘
```

Schema name
- Container

Schema signature
- contents + capacity
- Container is a set containing tuples consisting of two integers qualified 'contents' and 'capacity'

Schema Predicate
- contents <= capacity
- constrains tuples in the set

Equivalent to:
Container = {(contents, capacity) ∈ (Integer x Integer) | contents ≤ capacity}

Specifying Data-model Constraints

```
Container
contents: Integer
capacity: Integer
---
contents ≤ capacity
```

- The indicator's light is on iff the reading is less than the danger-level

```
Indicator
light: {on, off}
reading: Integer
danger_level: Integer
---
light = on ⇔
  reading ≤ danger level
```

```
Storage_Tank
c: Container
i: Indicator
---
i.reading = c.contents
c.capacity = 5000
i.danger_level = 50
```

- The storage tank links the contents of its container to the reading of the indicator
- It "initializes" the capacity and danger-level of its Container and Indicator

Specifying Operations

```
Fill_OK
Δ Storage_Tank
amount?: Integer
---
contents + amount? ≤ capacity
contents' = contents + amount?
```

- The Δ (greek letter delta) before Storage_Tank indicates that the values of the state variables will be changed
- The ? decorator after amount indicates that this is an input variable
- The first predicate acts as a precondition
- ... for the second predicate acting as postcondition

```
OverFill
Ξ Storage_Tank
amount?: Integer
---
r!: seq CHAR
contents + amount? > capacity
r! = "Insufficient Capacity"
```

- The Ξ (greek letter XI) before Storage_Tank indicates that the values of the state variables will not be changed
- The ! decorator after r indicates that this is an output variable

```
Fill
Fill_OK / Overfill
```

- The operation Fill is the logical or of the operations Fill_OK and OverFill

Firing Transitions

```
Storage_Tank
Δ Storage_Tank
amount?: Integer
---
contents - amount? ≥ 0
contents' = contents - amount?
```

```
s: Storage_Tank
s.contents = 3000       ... light = of
Dispense (s, 2999)      ... light = on
Fill (s, 500)   ... light = off
```

```
Container
contents: Integer
capacity: Integer
---
contents ≤ capacity
```

```
Storage_Tank
c: Container
i: Indicator
---
i.reading = c.contents
c.capacity = 5000
i.danger_level = 50
```

```
Indicator
light: {on, off}
reading: Integer
danger_level: Integer
---
light = on ⇔
  reading ≤ danger level
```


Consistent / Complete / Unambiguous

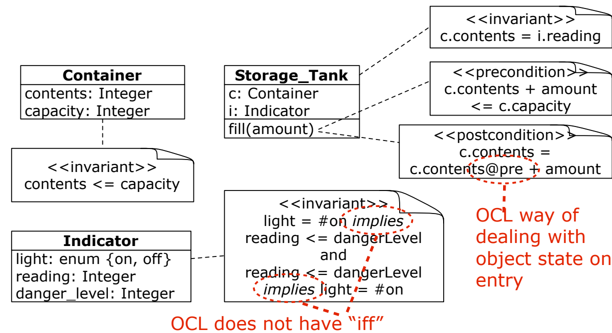
When is a logic-based specifcation ...

- Consistent
  + no contradictions in the specification (never "true=false")
- Complete
  + all properties are defined in terms of known concepts
  (all boils down to primitive domains Integer, Booleans, ...)
- Unambiguous
  + misinterpretations are impossible

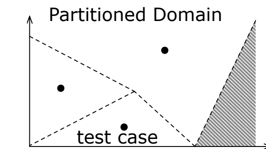
  + ... all done by means of logic deduction
  + In practice rarely applied due to the complexity of the models

Towards Design by Contract

- schemas become classes
- constraints on schemas become class invariants
- constraints on operations become pre- or post-conditions



Towards Testing



Test logic-based specification ?

- Equivalence partitioning
  based on Condition Testing

1. For each operation o in class C "flatten" applicable constraints (incl. superclass)

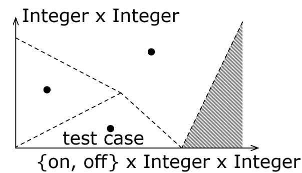
- flat_pre(o) := invariant(c) and pre(o)
- flat_post(o) := invariant(c) and post(o)
  - ➡ each operation o has boolean expression for pre- and postcondition
  - ➡ input/output domain for each method m is specified

2. For each operation o, cover domain as specified by flat_pre(o) and flat_post(o)

- Minimal Coverage: falsify each flat_pre(o) at least once
  + satisfy each flat_post(o) at least once
- Domain Testing: all combinations of true/false + almost "true/false"

Example: Fill

flat_pre(fill) = (c.contents = i.reading) and
(c.contents+amount <= c.capacity)
flat_post(o) = (c.contents = i.reading) and
(c.contents = c.contents@pre + amount)



c.contents = 5 i.reading = 5 c.capacity = 10	c.contents = i.reading	c.content + amount <= c.capacity	c.contents = c.contents@pre + amount	RESULT
amount = 3	true (=)	true (<)	true (=)	c.contents = 8
amount = 5	true (=)	true (=)	true (=)	c.contents = 10
amount = 7	true (=)	false (>)	false (<)	preconditionExc
$impossible	false (<)	--	--	--
$impossible	true (=)	--	--	--
$impossible	--	--	false (>)	--

Logic-Based Specifications Revisited

Logic-based Specifications

- Are particularly suitable for specifying constraints on class models (cf. OCL in UML)
  - ➡ constraints on schemas and operations
  - ➡ Design by Contract
- Specify operations as functions on sets
  - ➡ sets can be partitioned according to specification
  - ➡ Domain Testing

D) State-Based Specifications

Typically based on the notion of finite state machines

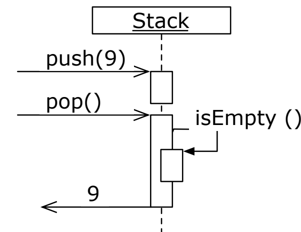
- describes the sequence of states a system is supposed to go through … in response to external stimuli (a.k.a. events)

StateCharts

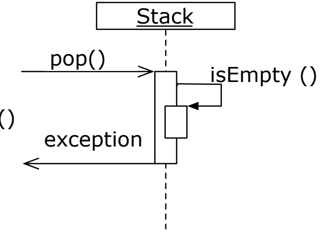
- widely used: present in UML
- commonly used in real-time systems
- Definitions
  - + state = a condition an object satisfies (i.e. a predicate computing its result with the attribute values of the object)
    - ➡ The state can be observed from the outside !
  - + transition = a change of state triggered by an event, condition or time

Sequence Diagrams

a) happy day scenario: pop of a non-empty stack



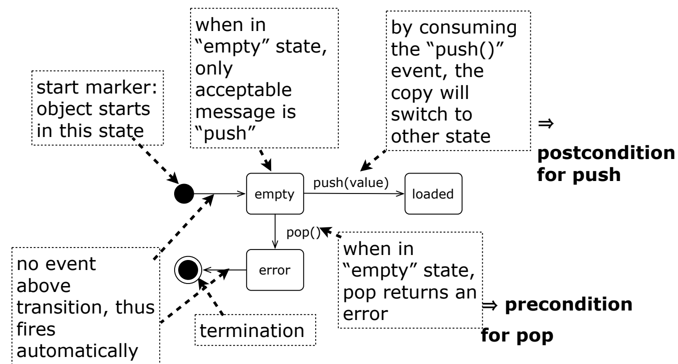
b) secondary scenario: pop of an empty stack



- What are acceptable message sequences for a stack ?
- What is the union of all possible scenario's ?

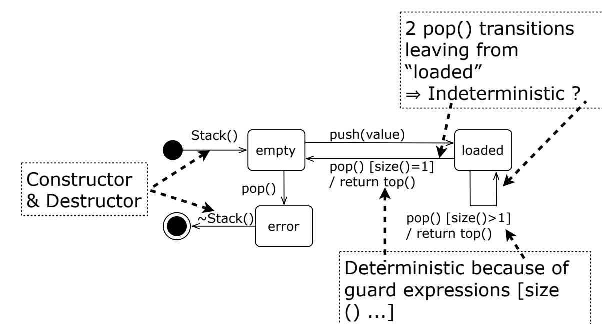
⇒ A state-chart allows to specify all valid (also all illegal) scenario's

State Chart for a Stack



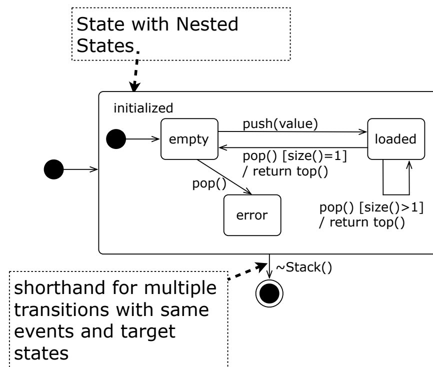
What about a pop() on a loaded stack ?

Guarded Transition



What about destructors on empty/loaded state ?

Nested State



Consistent / Complete / Unambiguous

When is a state-based specifcation ...

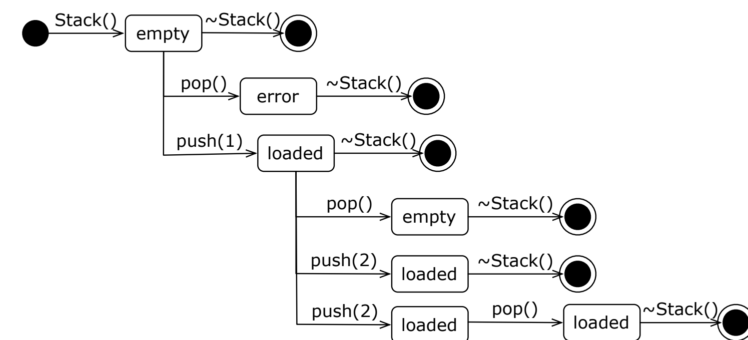
- Complete
  + every event/state pair has a transition
    - Create table: events (incl. guards) x state
      one cell contains target state
    - all cells should have a target state
- Consistent
  + every state is reachable from initial state
    & final state is reachable from every other state
    - Breadth-first spanning tree; root is initial state
    - all leaf nodes of the graph should be terminal state
- Unambiguous (= deterministic)
  + same event (incl. guard) does not appear on more than one transition leaving any given state
    - Verify using table created in completeness

Test Statecharts

Test cases
  + cover all state transitions
- define a predicate for each state,
  + which answers whether object is in that state
- test-cases must cover the breadth-first spanning tree
  + construct with same table used to verify completeness
    - rows and columns = events (incl. guards) x state
      cell contains target state

Example: Statechart "Stack"

	empty	loaded	error
Stack()	terminated	terminated	terminated
push	loaded	loaded	error (??)
pop() [size()=1]	error	empty	error (??)
pop() [size()>1]	error	loaded	error (??)



State-based Specifications Revisited

State-based Specifications

- Are particularly suitable for specifying “acceptable” message sequences
  - + unify effect of all possible scenarios on one class in one statechart
  - + “unacceptable” implies precondition
  - + state change implies postconditions
    - ➡ Design by Contract
- Specify acceptable message sequences as paths through a graph
  - ➡ cover all paths
  - ➡ Path Testing

Cleanroom Development

Are formal specifications useful in normal program development?

- Cleanroom method relies explicitly on formal specification
- Several projects have shown: it is cost and time effective to
  - + eliminate errors in specification and design
  - + ensure that implementation does not introduce defects

What is cleanroom development?

- Cleanroom Metaphor: Room to manufactures hardware(IC’s, ...)
  - ➡ Special measures are taken to avoid that dust enters the room
- Cleanroom development.
  - + advocates an incremental process model
  - + typically used with functional decomposition
  - + relies heavily on formal specification and verification (instead of unit testing)
  - + relies on code reviews to assure code quality
  - + relies on statistical testing for integration of increments

Correctness & Traceability

Correctness

- Are we building the system right?
  - + Formal specifications allow to verify presence of desired properties
    - ➡ Semi-automatic generation of test-cases

- Are we building the right system?
  - + (Some) formal verifications can be animated
    - ➡ May play the role of a prototype

Traceability

- Requirements ⇔ System?
  - + Formal specification is an intermediate representation
    - ➡ Traceability depends on usage and discipline

Summary(i)

You should know the answers to these questions

- What does it mean when we say that a formal specification is (a) consistent, (b) complete, and (c) unambiguous ?
- What does it mean for a state-chart to be (a) consistent, (b) complete, and (c) unambiguous?
- What does it mean for a algebraic specification to be (a) consistent and (b) complete?
- Why is an UML class diagram a semi-formal specification?
- Can you give 3 arguments against formal methods? Can you counter them?
- What’s the distinction between a semi-formal and a formal specification?
- In the code of the binary-search routine. If we replace the < in line 10 with an >, can you prove what is wrong with the program?
- In the discussion of Consistent / Complete / Unambiguous on page 22, can you verify that the addition of the axiom implies inconsistency? And can you show that the removal of the axiom implies incompleteness?

You should be able to complete the following tasks

- Extend the Z specification of the storage tank to include a “full” indicator, indicating when a Storage_Tank is close to capacity.

Summary(ii)

You should be able to complete the following tasks (cntnd.)

- Provide the sequence charts for the primary and secondary scenarios of the "Place Order" use case in the chapter on use case. Derive the corresponding state-chart for the class "Order".
- Given a Z or OCL specification, derive a test model using condition testing.
- Given a statechart specification, derive a test model using path testing.

Can you answer the following questions?

- Why is it likely that you will encounter formal specifications?
- Explain the relationship between "Design By Contract" on the one hand and "Input/Output Specifications", "Algebraic Specifications", "Logic-Based Specifications" and "State based specifications" on the other hand.
- Explain the relationship between "Testing" on the one hand and "Logic-Based Specifications" and "State based specifications" on the other hand.
- Why is it necessary to complement sequence diagrams with state charts?
- What does Cleanroom Development have to do with Formal Specifications?
- You are supposed to build an e-commerce system for selling books over the world-wide web. You must guarantee 24 hours operations and secure transactions. Your boss asks you to look into formal specs; which ones would you advise and why?