CS 4390/5387 SOFTWARE V&V

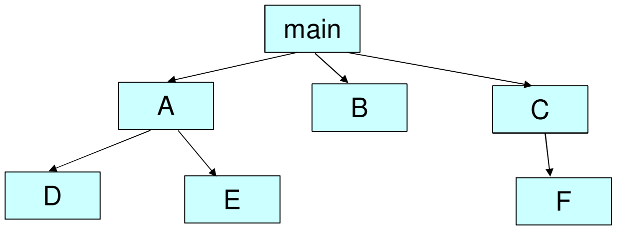
LECTURE 8
INTEGRATION TESTING

Outline

- Integration Testing
  - Definition
  - Strategies
    - Big bang
    - Top-down
    - Bottom-up
    - Sandwich
- Integrating OO Applications

Integration testing

- Done between unit testing and system testing
- Objective: ensure assembled modules, that work fine in isolation, work well together
  - Attempt to find interface faults
- Need a module call graph.



$$\begin{tikzcd}
 & & \text{main} \arrow[dl] \arrow[d] \arrow[dr] & \\
 & A \arrow[dl] \arrow[d] & B & C \arrow[d] \\
D & E & & F
\end{tikzcd}$$

Integration Strategy

- How low-level modules are assembled to form higher-level program entities
- Strategy impacts
  - the type of test tools to use
  - the order of coding/testing units
  - the cost of generating test cases
  - the cost of locating and correcting detected defects

Integration testing strategies

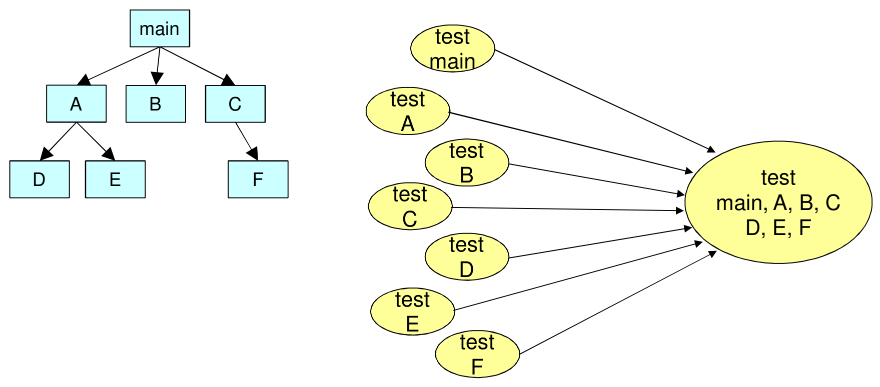
- Several different strategies can be used for integration testing

- Comparison criteria:
  - fault localization
  - effort needed (for stubs and drivers)
  - degree of testing of modules achieved
  - possibility for parallel development

- Examples of strategies
  - Big-bang
  - Top-down
  - Bottom-up
  - Sandwich

Big Bang Integration - 1

- Non-incremental strategy
  - Unit test each module in isolation
  - Integrate as a whole



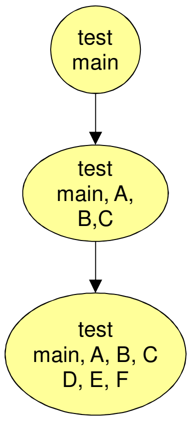
Big Bang Integration - 2

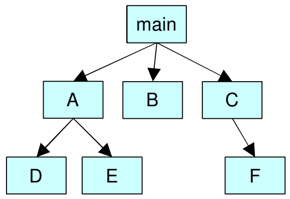
- Advantages
  - Convenient for small systems

- Disadvantages
  - Integration testing can only begin when all modules are ready
  - Fault localization difficult
  - Easy to miss interface faults

Top-down Integration - 1

- Incremental strategy
  1. Start by including highest level modules in test set.
     - All other modules replaced by stubs or mock objects.
  2. Integrate (i.e. replace stub by real module) modules called by modules in test set.
  3. Repeat until all modules in test set are integrated. ***This includes regression testing***

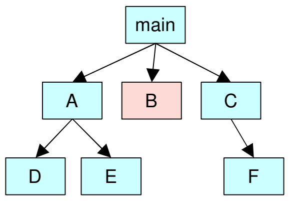


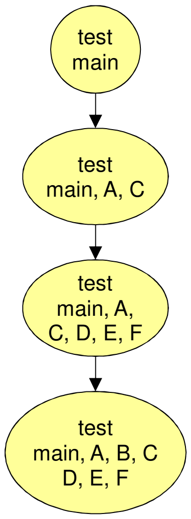


Top-down Integration - 2

9

- The integration order can be modified to:
  - include critical modules first
  - leave modules not ready to later





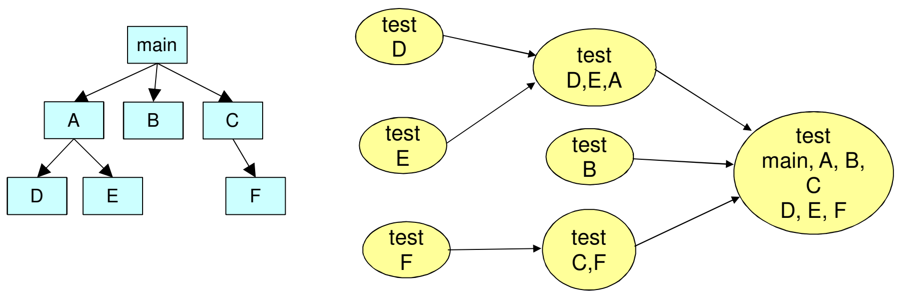
Top-down Integration - 3

10

- Advantages
  - Fault localization easier
  - Few or no drivers needed
  - Possibility to obtain an early prototype
  - Different order of testing/implementation possible
  - Major design flaws found first
    - in logic modules on top of the hierarchy
- Disadvantages
  - Need lot of stubs / mock objects
  - Potentially reusable modules (in bottom of the hierarchy) can be inadequately tested (Why?)

Bottom-up Integration - 1

- Incremental strategy
  - Test low-level modules, then
  - Modules calling them until highest level module



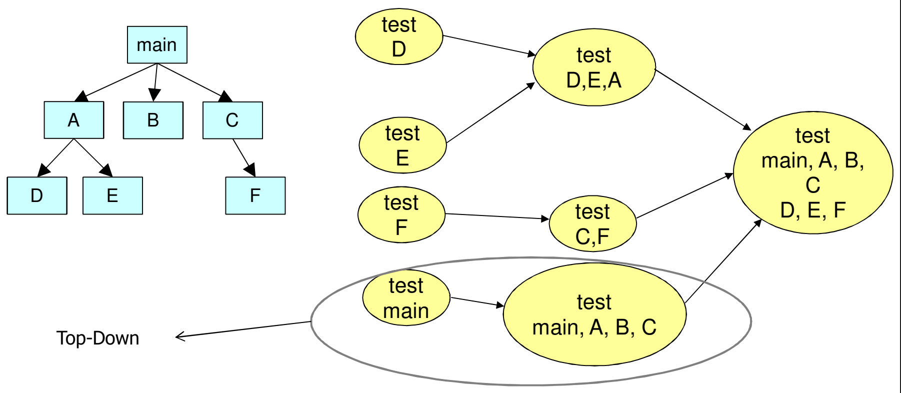
Bottom-up Integration - 2

- Advantages
  - Fault localization easier (than big-bang)
  - No need for stubs / fewer mock objects
  - Operational modules tested thoroughly
  - Testing can be in parallel with implementation
- Disadvantages
  - Need drivers
  - High-level modules (that relate to the solution logic) tested in the last (and least)

Sandwich Integration

13

- Combines top-down and bottom-up approaches
- Distinguish 3 layers
  - logic (top) - tested top-down
  - middle
  - operational (bottom) – tested bottom-up



Risk Driven integration

14

- Integrate base on criticality
  - most critical or complex modules integrated first with modules called

Integration of Object-Oriented Systems

- Different scopes
  - Intra Class (methods within a class)
    - Big-bang
    - Bottom-up: constructors - accessors - predicates– iterators – destructors
    - State based integration
  - Cluster Integration
    - Big Bang
    - Bottom-up
    - Top-down
    - Scenario based
  - Subsystem/System Integration
    - Big Bang
    - Bottom-up
    - Top-down
    - Scenario based

Intra-Class Integration

- Operations in classes (methods)
  - black box, white box approaches
  - each exception raised at least once
  - each interrupt forced to occur at least once
  - each attribute set interrogated at least once
  - state-based testing (Following Slides)

- Integration of methods
  - Big-bang used for situation where methods are tightly coupled
  - complex methods can be tested with stubs/mocks

State-Based Testing

17

- Natural representation with finite state machines
  - States correspond to certain values of the attributes
  - Transitions correspond to method calls
- FSM can be used as basis for testing
  - e.g. “drive” the class through all transitions, and
  - verify the response and the resulting state

FSM Example: Stack

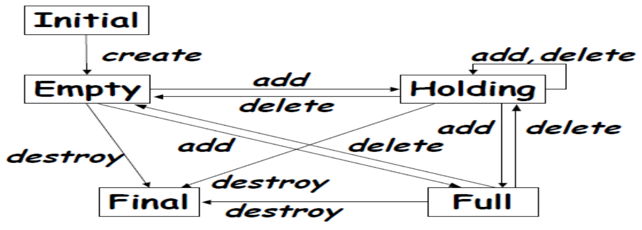
18

- States
  - Initial: before creation
  - Empty: number of elements = 0
  - Holding: number of elements >0, but less than the max capacity
  - Full: number elements = max capacity
  - Final: after destruction
- Transitions: starting state, ending state, event that triggers the transition, and possibly some response to the event

Possible Transitions

19

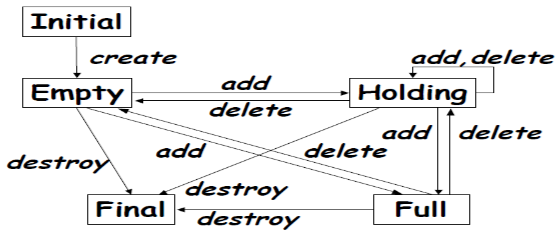
- Initial → Empty: event= "create"
  - e.g. "s = new Stack()" in Java
- Empty → Holding: event = "add"
- Empty → Full: event = "add"
  - if max_capacity = 1
- Empty → Final: event = "destroy"
  - e.g. destructor call in C++, garbage collection in Java
- Holding → Empty: event = "delete"



FSM-based Testing

20

- Each valid transition should be tested
  - Verify the resulting state using a state inspector that has access to the internals of the class
- Each invalid transition should be tested to ensure that it is rejected and the state does not change
- e.g. Full → Full is not allowed (Example?):
  - Need to test calling add on a full stack



Exercise

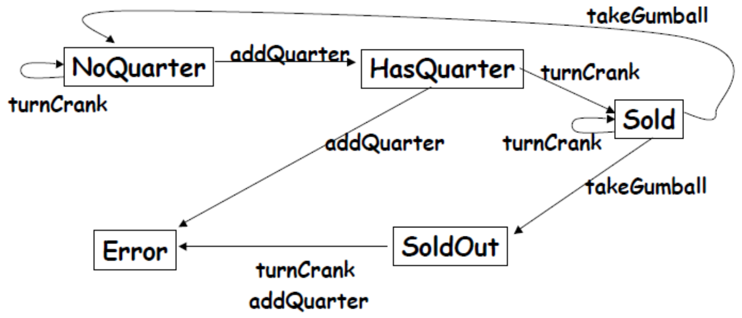
21

- Gumball machine example
- States:
  - NoQuarter, HasQuarter, Sold, SoldOut, Error
- Transitions:
  - turnCrank(), addQuarter(), takeGumball()

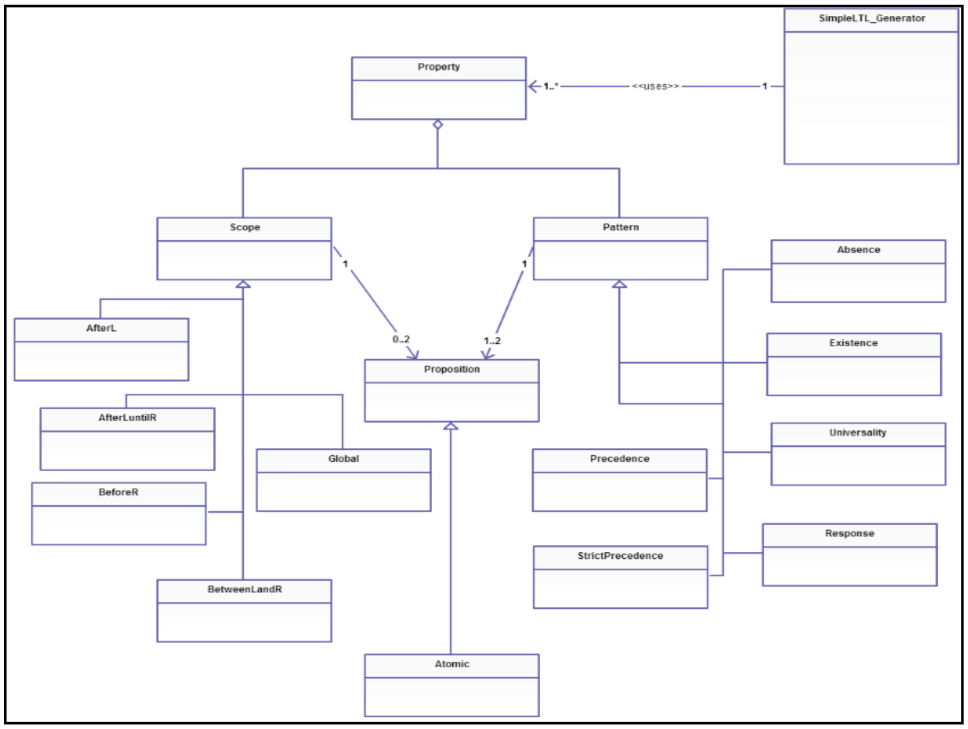
(you may add states/transitions if necessary)

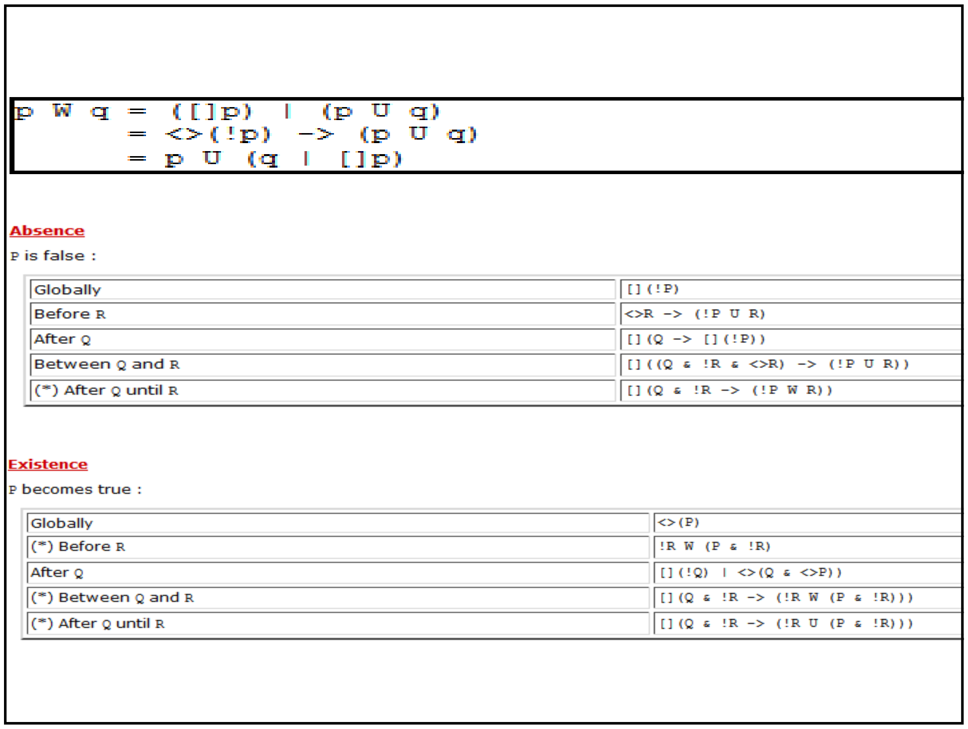
Solution

22



Possible Tests?





```
p W q = ([]p) | (p U q)
      = <>(!p) -> (p U q)
      = p U (q | []p)
```

Absence

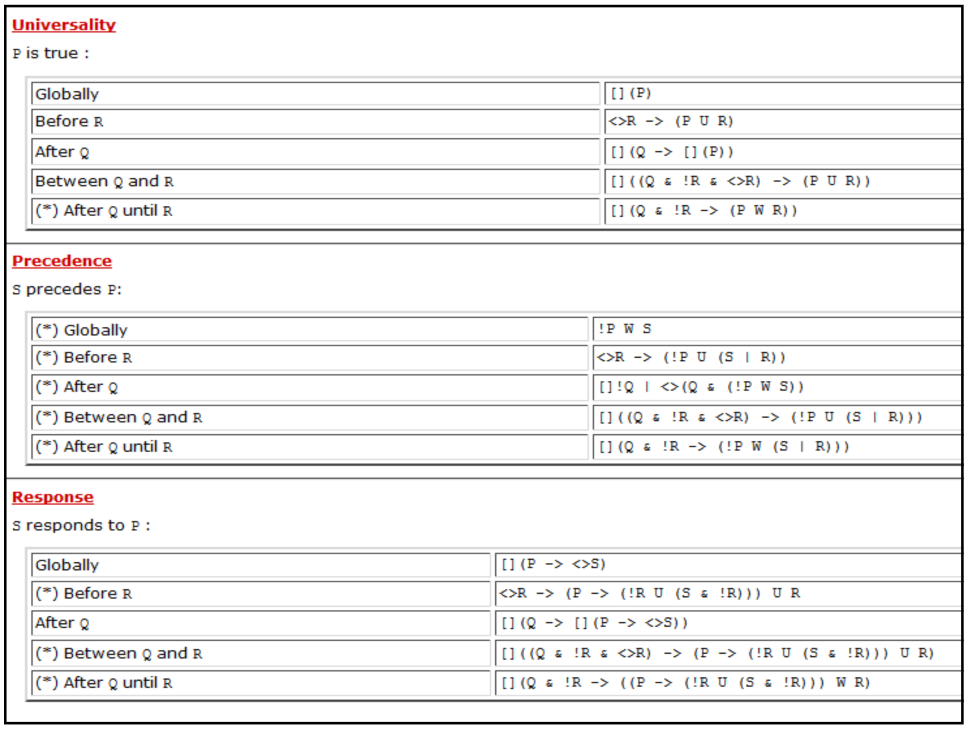
P is false :

Globally	`[](!P)`
Before R	`<>R -> (!P U R)`
After Q	`[](Q -> [](!P))`
Between Q and R	`[]((Q & !R & <>R) -> (!P U R))`
(*) After Q until R	`[](Q & !R -> (!P W R))`

Existence

P becomes true :

Globally	`<>(P)`
(*) Before R	`!R W (P & !R)`
After Q	`[](!Q) | <>(Q & <>P))`
(*) Between Q and R	`[](Q & !R -> (!R W (P & !R)))`
(*) After Q until R	`[](Q & !R -> (!R U (P & !R)))`



Universality

P is true :

Globally	`[](P)`
Before R	`<>R -> (P U R)`
After Q	`[](Q -> [](P))`
Between Q and R	`[]((Q & !R & <>R) -> (P U R))`
(*) After Q until R	`[](Q & !R -> (P W R))`

Precedence

S precedes P:

(*) Globally	`!P W S`
(*) Before R	`<>R -> (!P U (S | R))`
(*) After Q	`[]!Q | <>(Q & (!P W S))`
(*) Between Q and R	`[]((Q & !R & <>R) -> (!P U (S | R)))`
(*) After Q until R	`[](Q & !R -> (!P W (S | R)))`

Response

S responds to P :

Globally	`[](P -> <>S)`
(*) Before R	`<>R -> (P -> (!R U (S & !R))) U R`
After Q	`[](Q -> [](P -> <>S))`
(*) Between Q and R	`[]((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)`
(*) After Q until R	`[](Q & !R -> ((P -> (!R U (S & !R))) W R)`

Team Assignment

- Develop an integration test plan for the Simple LTL Generator system on slide #23
  - Order of classes to be integrated
  - Integration tests for each module (group of modules) at each step
  - Necessary stubs/drivers.
  - Due on…