Integration Testing

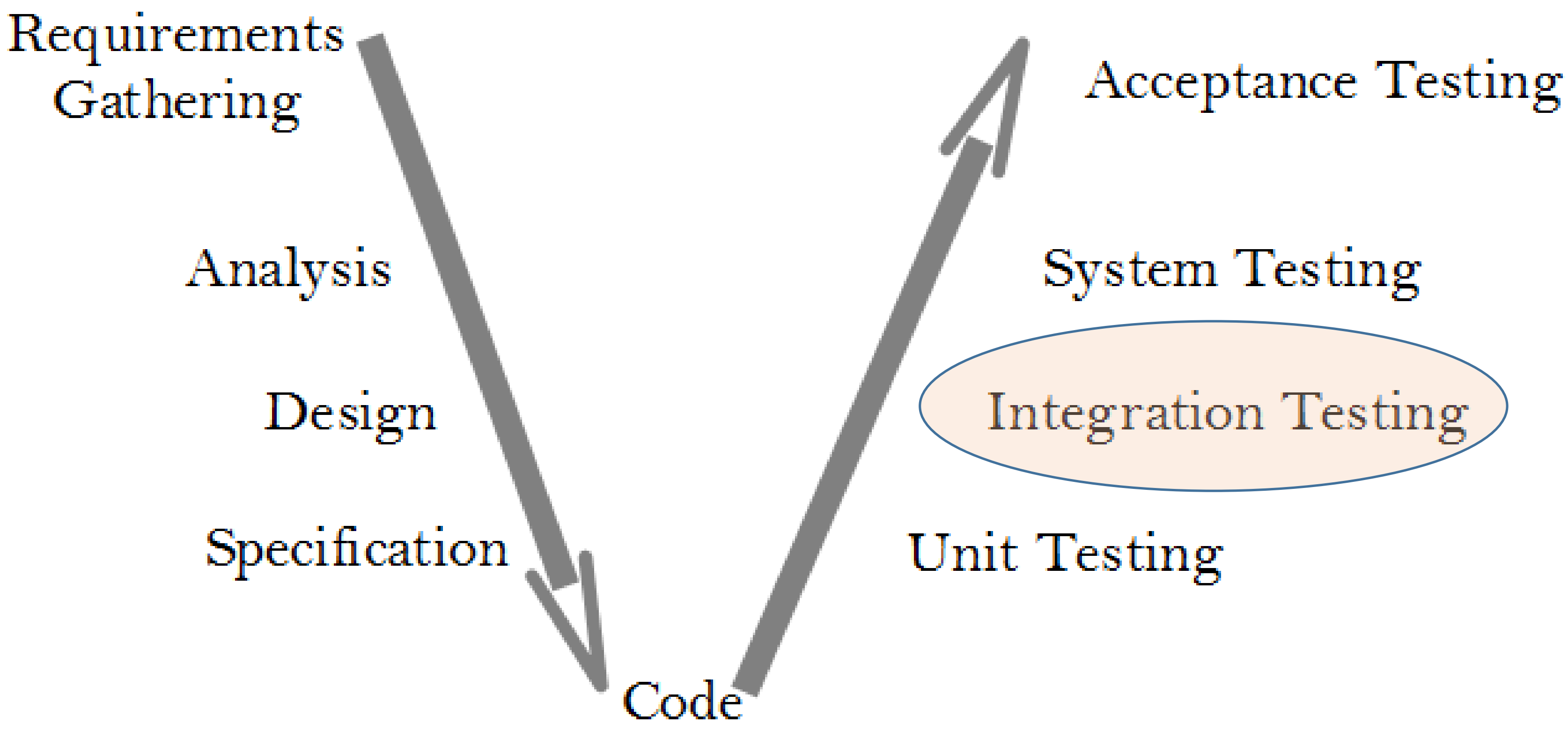
Spring, 2023

Yi Xiang xiangyi@scut.edu.cn

Contents

- What is Integration Testing
- Approaches to Integration Testing
  - Decomposition-based Integration
  - Call-Graph based Integration
  - MM Path based Integration

1. Lifecycle Testing Relationships



Testing Level Assumptions and Objectives

- Unit assumptions
  - All other units are correct
  - Compiles correctly
- Integration assumptions
  - Unit testing complete
- System assumptions
  - Integration testing complete
  - Tests occur at port boundary

- Unit goals
  - Correct unit function
  - Coverage metrics satisfied
- Integration goals
  - Interfaces correct
  - Correct function across units
  - Fault isolation support
- System goals
  - Correct system functions
  - Non-functional requirements tested
  - Customer satisfaction.

Definitions – Integration Tests

- Integration test data is selected to ensure that the components or sub-systems of a system are working correctly together.

- Test cases will explore different interactions between the components, and make sure the correct results are produced.

Definitions – System Tests

- System test data is selected to ensure that the system as a whole is working.

- Test cases will therefore explore the different inputs and combinations of inputs to the system to ensure that the system satisfies its specification

The Mars Climate Orbiter Mission

- mission failed in September 1999
  - completed successful flight: 416,000,000 miles (665.600.600 km)
  - 41 weeks flight duration
  - lost at beginning of Mars orbit

- An integration fault:
  - Lockheed Martin used English units for acceleration calculations (pounds), and Jet Propulsion Laboratory used metric units (newtons).

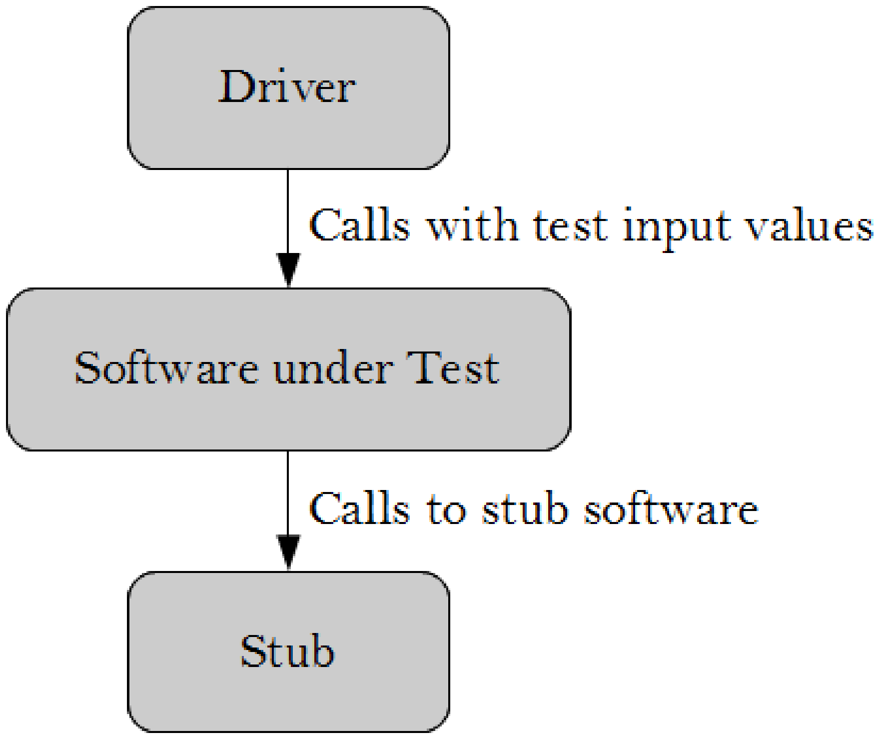
- NASA announced a US$ 50,000 project to discover how this happened.

Integration Testing – Drivers and Stubs

- Drivers and Stubs are temporary software components

- A test driver calls the software under test, passing the test data as inputs.

- In manual testing, where the system interface has not been completed, a test driver is used in its place to provide the interface between the test user and the software under test.



Integration Testing – Drivers and Stubs

➢ Drivers

- Drivers can have varying levels of sophistication.
- It could be hard-coded to run through a fixed series of input values, read data from a prepared file, contain a suitable random number generator etc..

➢ Stubs

- A stub is a temporary or dummy software that is required by the software under test to operate properly.
- This is a throw-away version to allow testing to take place.
- It will provide a fixed or limited set of values to be passed to the software under test.

2. Approaches to Integration Testing (“source” of test cases)

- **Decomposition-based Integration**
  - “Big bang” integration
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- **Call graph-based Integration**
  - Pairwise integration
  - Neighborhood integration
- **Path-based Integration**
  - MM-Path based Integration

2.1 Decomposition-based Integration

- In this strategy, do the decomposition based on the functional characteristics of the system.
  - A functional characteristic is defined by what the module does, that is, actions or activities performed by the module.

Big bang integration

- big-bang groups the whole system and test it in a single test phase.

Top-down integration

- Top-down starts at the root of the tree and slowly work to lower level of the tree

Bottom-up integration

- Bottom-up mirrors top-down, it starts at the lower level implementation of the system and work towards the main

Sandwich integration

- Sandwich is an approach that combines both top-down and bottom-up.

Big bang Integration

- Considers the whole system as a subsystem
- Tests all the modules in a single test session
- Only one integration testing session

No...
– stubs
– drivers
– strategy

- Very difficult fault isolation

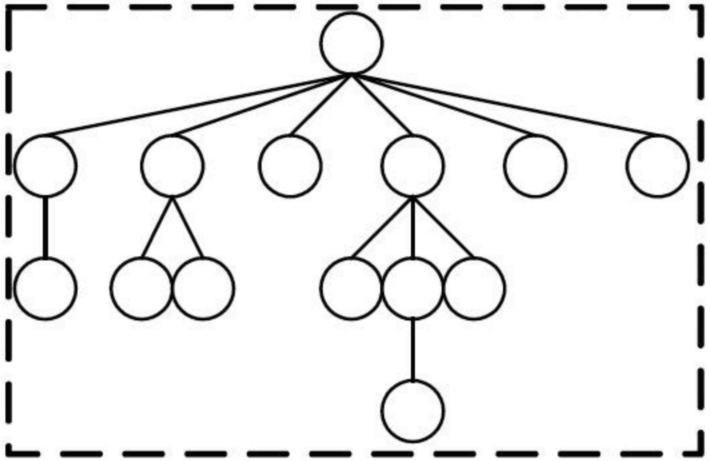


Figure 1.1: Big bang integration, coverage of a test session.

Top-Down Integration

- **Breadth-first traversal** of the functional decomposition tree.
  - First step: Check main program logic, with all called units replaced by stubs that always return correct values.
  - Move down one level
    - – replace one stub at a time with actual code.
    - – any fault must be in the newly integrated unit

- Early SUT prototype

- Throw-away code programming

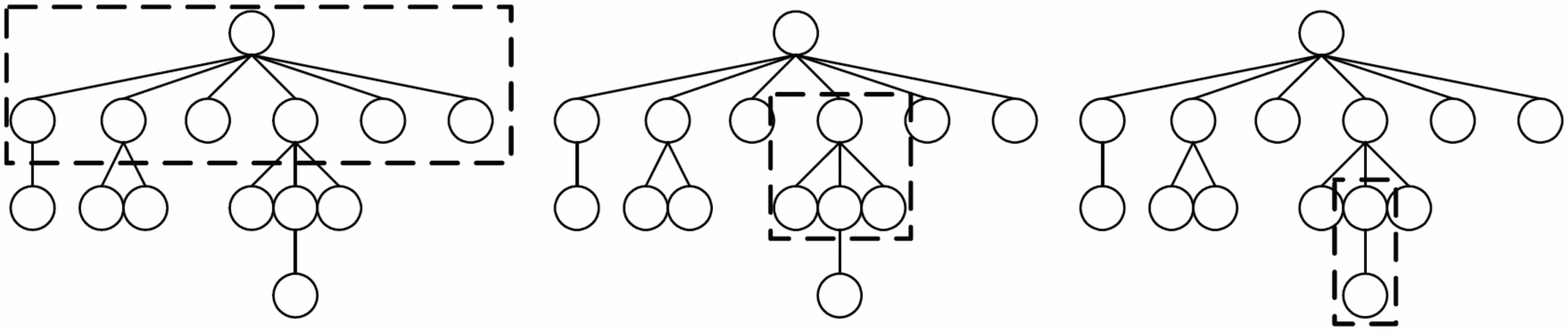


Figure 1.3: Top-down integration, coverage of different sessions at different levels.

Bottom-Up Integration

- Reverse of top-down integration
  - Start at leaves of the functional decomposition tree.
  - Driver units...
    - – call next level unit
    - – “drive” the unit with inputs
- As with top-down integration, one driver unit at a time is replaced with actual code.
- Any fault is (most likely) in the newly integrated code.
- Less throw-away code programming
- No prototype and Main program tested last

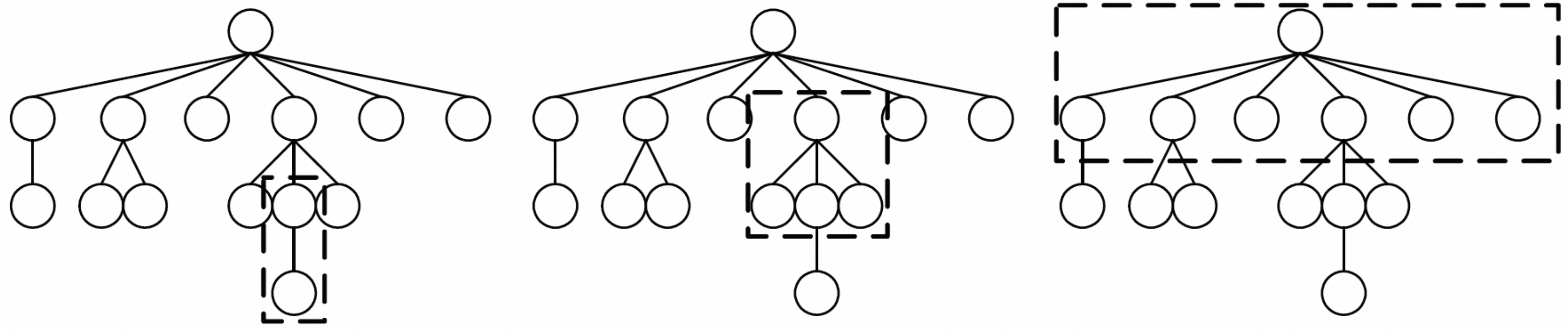


Figure 1.5: Bottom-up integration, coverage of different sessions at different levels.

Sandwich integration

- Combines top-down approach and bottom-up approach
  - Generally, higher level modules use a top-down approach (stub)
  - Normally, lower-level modules use a bottom-up approach (driver)
- Testing converges to the middle
- Number of integration sessions can vary

- Top and bottom layers can be done in parallel
- Less stubs and drivers needed

- Hard to isolate problems

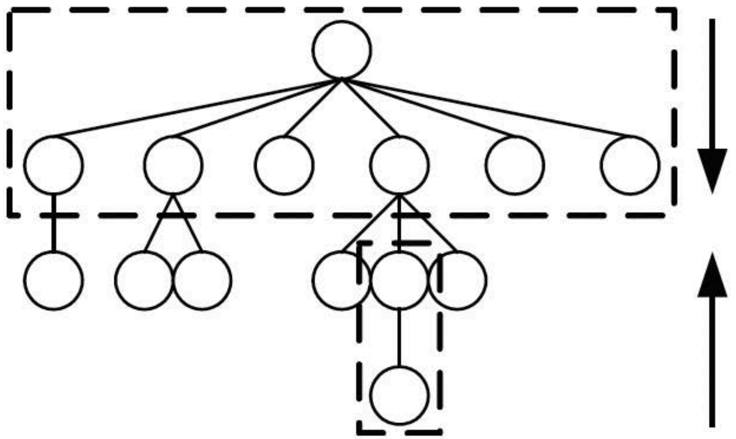


Figure 1.7: Sandwich integration, combining top-down and bottom-up approach.

CaseStudy --- The NextDate Program

- This program uses three variables: month, date and year.
- With the input, it returns the next date of the inputted date.

It has the following characteristics:

- Checks for valid input:
  - Must be positive integers
  - A month must be: $1 \leq month \leq 12$
  - A day must be: $1 \leq day \leq 31$
  - A year must be: $1812 \leq 2012$
- Takes leap years into consideration:
  - A year is leap year if is divisible by 4
  - Unless it is a century year, then it is only leap if is multiple of 400

Pseudo-code of the NextDate program implementation- Main()

```
input        : A date
output       : Next date of the day inputted

Type Date: Integer month,Integer day,Integer year
Date         : today,tomorrow,aDate

Main integrationNextDate
    getDate(today);                          /* message 1 */
    printDate(today);                        /* message 2 */
    tomorrow = incrementDate(today);         /* message 3 */
    printDate(tomorrow);                     /* message 4 */
End Main
```

Function (*Main*) integrationNextDate(date)

Pseudo-code of the NextDate program implementation – isLeap()

```
input  : A year
output: True if is a leap year, otherwise false

Function Boolean isLeap(year)
    if year divisible by 4 then
        if year is NOT divisible by 100 then  isLeap = True
        else if year is divisible by 400 then  isLeap = True
        else  isLeap = False
    else  isLeap = False
End (Function isLeap)
```

Function `isLeap`(*year*)

Pseudo-code of the NextDate program implementation – lastDayOfMonth()

```
input  : A month and a year
output: The last day of the month in integer

Function Integer lastDayOfMonth(month,year)
    switch month do
        case 1,3,5,7,8,10,12:  lastDayOfMonth = 31
        case 4,6,9,11:  lastDayOfMonth = 30
        case 2:
            if isLeap(year) then  lastDayOfMonth = 29;          /* message 5 */
            else  lastDayOfMonth = 28
        endsw
    endsw
End (Function lastDayOfMonth)
```

Function `lastDayOfMonth(`*month,year*`)`

Pseudo-code of the NextDate program implementation – validDate()

```
input  : A date
output: True if the date is valid, otherwise false

Function Boolean validDate(aDate)
    if (aDate.Month > 0) ∧ (aDate.Month <= 12) then monthOK = True
    else monthOK = False
    if monthOK then
        if (aDate.Day > 0) ∧ (aDate.Day <= lastDayOfMonth(aDate.Month,aDate.Year)) then dayOK =
        True;                                                    /* message 6 */
        else dayOK = False
    endif
    if (aDate.Year > 1811) ∧ (aDate.Year <= 2012) then yearOK = True
    else yearOK = False
    if monthOK ∧ dayOK ∧ yearOK then validDate = True
    else validDate = False
End (Function validDate)
```

Function validDate(*aDate*)

Pseudo-code of the NextDate program implementation –getDate()

```
input  : A set of integers from user input
output: A date
Function Date getDate(aDate)
    repeat
        Output("Enter a month:")
        Input(aDate.Month)
        Output("Enter a day:")
        Input(aDate.Day)
        Output("Enter a year:")
        Input(aDate.Year)
        getDate.Month = aDate.Month
        getDate.Day = aDate.Day
        getDate.Year = aDate.Year
    until validDate(aDate);                                   /* message 7 */
End (Function getDate)
```

Function `getDate(`*aDate*`)`

Pseudocode of the NextDate program implementation – incrementDate()

```
input  : A date
output: The date incremented

Function Date incrementDate(aDate)
    if aDate.Day < lastDayOfMonth(aDate.Month) then  aDate.Day = aDate.Day + 1;      /* message 8 */
    else
        aDate.Day = 1
        if aDate.Month = 12 then
            aDate.Month = 1
            aDate.Year = aDate.Year + 1
        else  aDate.Month = aDate.Month + 1
    endif
End (Function incrementDate)
```

Function `incrementDate(`*aDate*`)`

Pseudo-code of the NextDate program implementation – printDate()

```
input  : A date
output: The date in string
Procedure String printDate(aDate)
    Output("Day is ",aDate.Month,"/",aDate.Day,"/",aDate.Year)
End (Procedure printDate)
```

Procedure printDate(*aDate*)

Decomposition-based Integration --- the NextDate program (Big Bang)

Compile all the modules in the functional decomposition tree and test the whole system in a single session

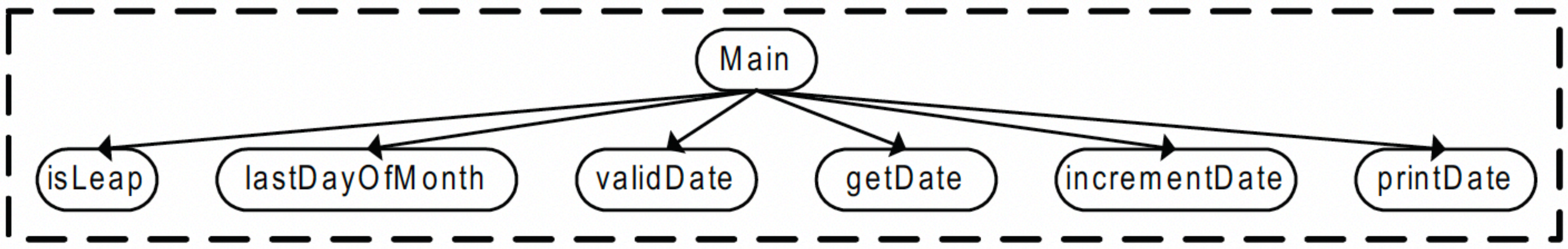


Figure 1.2: Big bang integration of the *nextDate* program.

Decomposition-based Integration --- the NextDate program (Top-down)

- Start with Main as a target node and replace the children nodes one by one with stubs (only one stub in each test session).
- We must build the stub such that it returns correct values to the real module and compatible to the test cases.

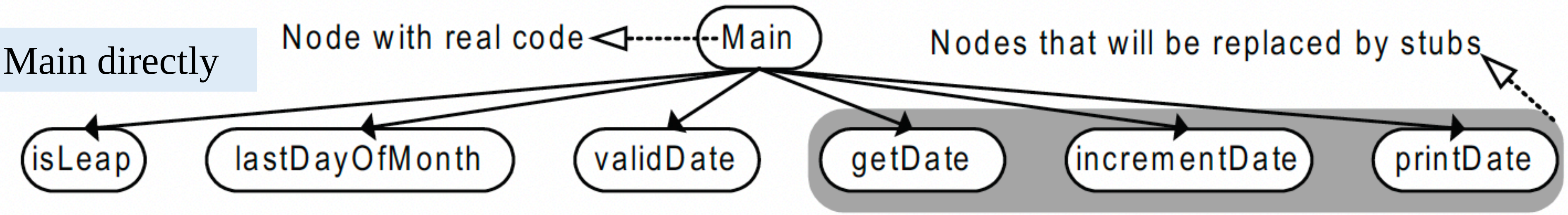
A possible stub for incrementDate:

```
input  : A date
output: The following day of the inputted date
Date   : d31121999,d28022000,d28021999,next

Function Date incrementDate(aDate)
    d12311999 = Date(12,31,1999)
    d02282000 = Date(02,28,2000)
    d02281999 = Date(02,28,1999)
    if aDate == d12311999 then  next = Date(01,01,2000)
    else if aDate == d02282000 then  next = Date(02,29,2000)
    else if aDate == d02281999 then  next = Date(03,01,1999)
End (Function incrementDate)
```

The test cases will be limited by how and what we code in the stub

Never called by Main directly



Not be able to isolate them withstubs with a top-down approach

Decomposition-based Integration --- the NextDate program (Bottom-up)

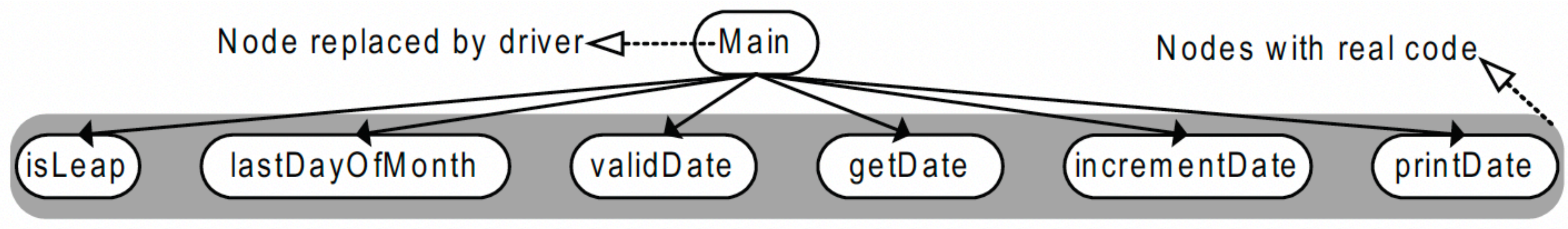
Begins with the leaves of the decomposition tree, and use a driver version of the unit that would normally call it to provide it with test cases.

- No need to substitute as many modules with temporary throw-away modules

A possible driver for isLeap:

```
input  : None (Hardcoded test cases)
output: True if it match the expected result, otherwise false

Function Integer  isLeap_driver()
    Output("Pass 1900: ",(isLeap(1900) == False))
    Output("Pass 1999: ",(isLeap(1999) == False))
    Output("Pass 2000: ",(isLeap(2000) == True))
End (Function  isLeap_driver)
```



Decomposition-based Integration --- the NextDate program (Sandwich)

Sandwich integration combines top-down integration and bottom-up integration.

- In top-down by starting at the root of the functional decomposition tree, which can test the main program at early stage.
- In bottom-up, we will have coverage that is easy to create test cases.

There are no strict guidelines in modules grouping

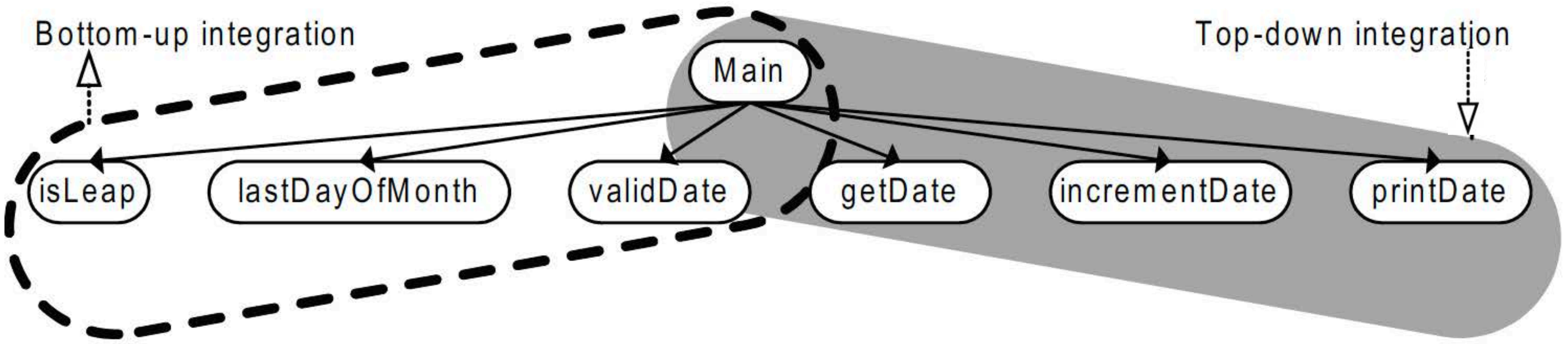


Figure 1.8: Sandwich integration of the *nextDate* program.

Pros and Cons of Decomposition-Based Integration

- Pros
  - – intuitively clear
  - – “build” with proven components
  - – fault isolation varies with the number of units being Integrated

- Cons
  - – some branches in a functional decomposition may not correspond with actual interfaces.
  - – stub and driver development can be extensive

2.2 Call Graph-based Integration

- Definition: The Call Graph of a program is a directed graph in which
  – nodes are unit
  – edges correspond to actual program calls (or messages)

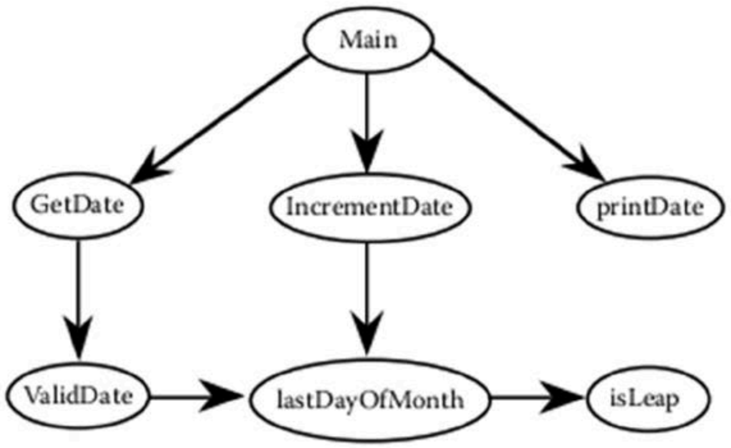
- Call Graph Integration avoids the possibility of impossible edges in decomposition-based integration.
- Can still use the notions of stubs and drivers.
- Can still traverse the Call Graph in a top-down or bottom-up strategy.

Two strategies

– Pair-wise integration
– Neighborhood integration

Degrees of nodes in the Call Graph indicate integration sessions

– test high indegree nodes first, or at least,
– pay special attention to "popular" nodes



Pair-Wise Integration

- By definition, and edge in the Call Graph refers to an interface between the units that are the endpoints of the edge.

- Every edge represents a pair of units to test.
- Fault isolation is localized to the pair being Integrated
- The number of integration testing sessions is the number of edges

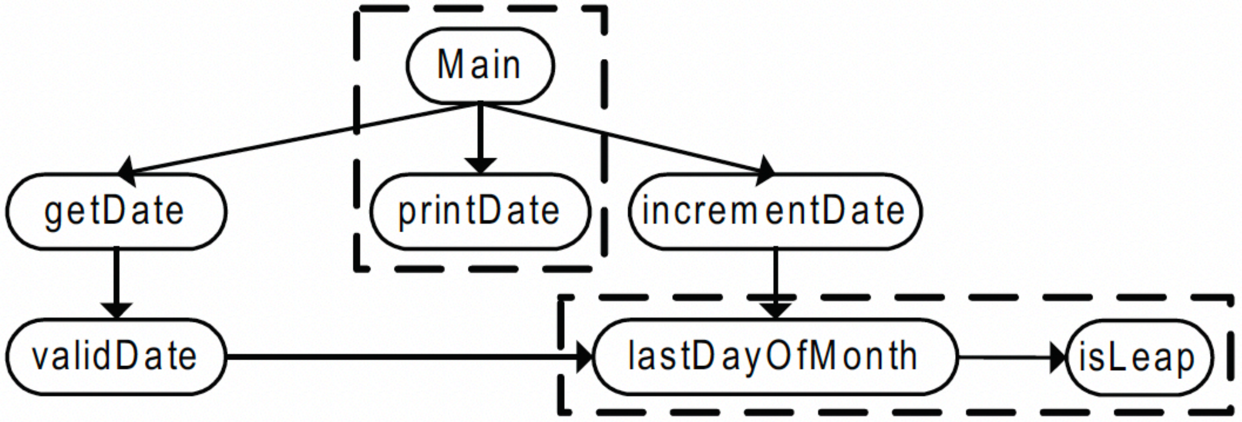


Figure 2.2: Two pairs used in pairwise integration of the *nextDate* program.

Neighborhood Integration

- The neighborhood (or radius 1) of a node in a graph is the set of nodes that are one edge away from the given node.

- This can be extended to larger sets by choosing larger values for the radius.
- Stub and driver effort is reduced.

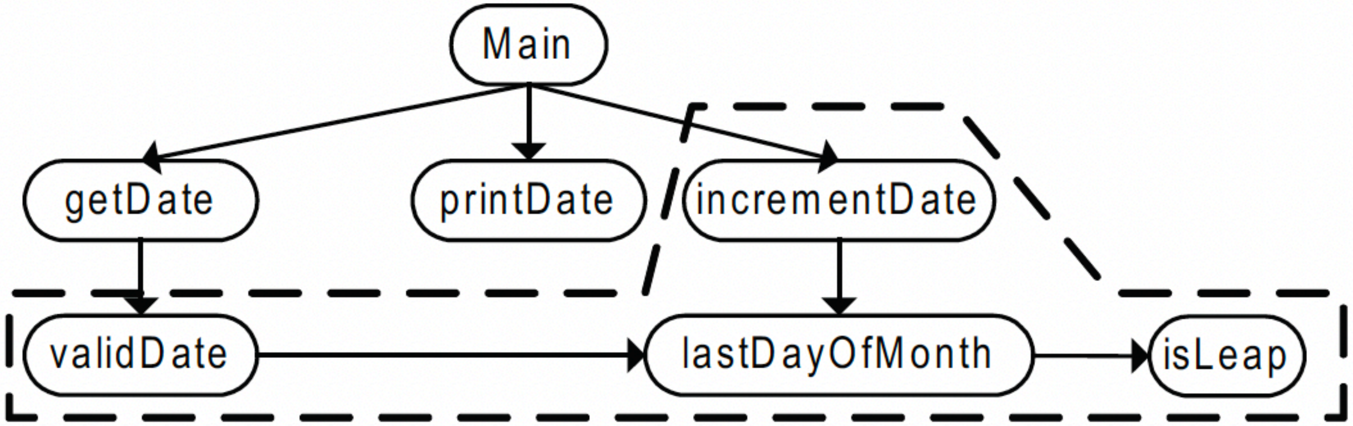


Figure 2.4: Neighbourhood of *lastDayOfMonth* in neighbourhood integration of the *nextDate* program.

2.3 Path-Based Integration

- Motivation : an integration testing level construct similar to Paths Coverage for unit testing

– extend the symbiosis of spec-based and code-based testing to the integration level
– greater emphasis on behavioral threads
– shift emphasis from interface testing to interactions (cofunctions) among units

- Need some new definitions
    - source and sink nodes in a program graph
    - module (unit) execution path
    - generalized message
    - MM-Path

New and Extended Definitions

- A source node in a program is a statement fragment at which program execution begins or resumes.
- A sink node in a unit is a statement fragment at which program execution terminates.
- A module execution path is a sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes.
- A message is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.
- Module/Message-Path – an interleaved sequence of module execution paths and messages.

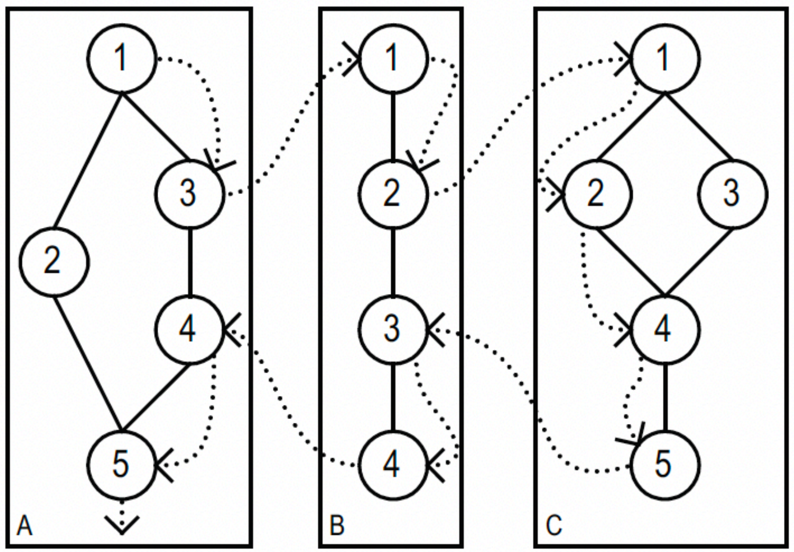
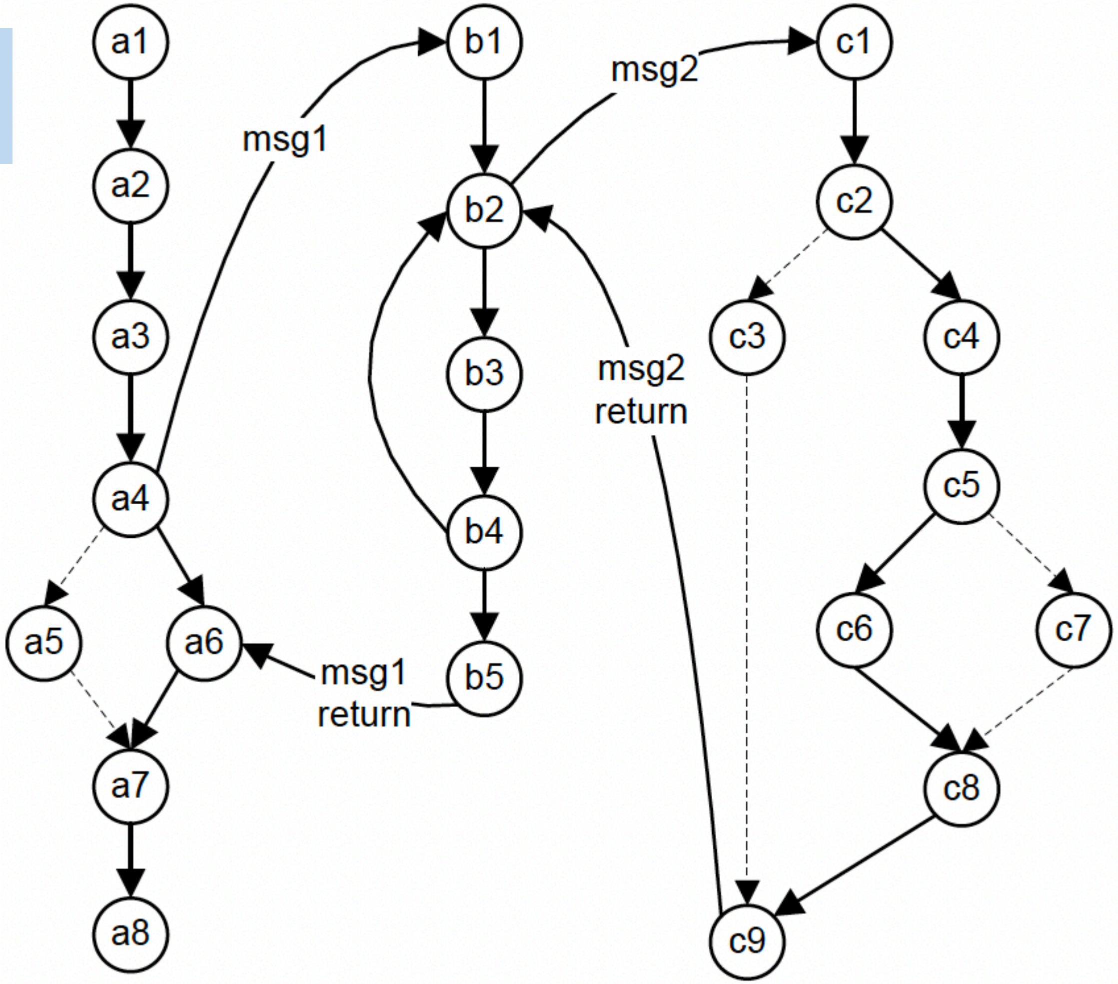


Figure 3.1: A MM-Path (dotted arrows) across units *A*, *B and C*.

MM-Path Definition and Example

An MM-Path is an interleaved sequence of module execution paths and messages.

An MM-Path across three units



The node sequence

<a1, a2, a3, a4>
message msg1
<b1, b2>
message msg2
<c1, c2, c4, c5, c6, c8, c9>
msg2 return
<b3, b4, (b2, b3, b4)*, b5>
msg1 return
<a6, a7, a8>

MM-Path based Integration --- the NextDate program

The MM-Paths begin in and return to the main program.

Main problem is knowing how many MM-Paths are required to complete the integration test. The set of MM-Paths should traverse all source-to-sink paths.

The following fragment represent the first MM-Path for “5/27/2002”

```
Main(1,2)
    message1
    getDate(36,37,38,39,40,41,42,43,44,45,46,47)
        message7
        validDate(24,25,26,27,28)
            message6
            lastDayOfMonth(14,15,16,23);            /* Point of quiescence */
        validDate(28,30,31,33,35)
    getDate(48)
Main(3)
```

Pros and Cons of Path-Based Integration

- Pros
  - Hybrid of functional and structural testing
  - Closely coupled with actual system behaviour
  - Does not require stub or driver
- Cons
  - Extra effort required to identify the MM-Paths

Comparison of Integration Testing Strategies

Strategy Basis	Ability to test interfaces	Ability to test co-functionality	Fault isolation and resolution
Functional Decomposition	acceptable, but can be deceptive (phantom edges)	limited to pairs of units	good to a faulty unit
Call Graph	acceptable	limited to pairs of units	good to a faulty unit
MM-Path	excellent	complete	excellent to a faulty unit execution path