Component Validation Methods - Black-Box Testing

San José State University

By Jerry Zeyu Gao, Ph.D.
San Jose State University
Email: jerrygao@email.sjsu.edu
URL:/www.engr.sjsu.edu/gaojerry

Component Validation Methods - Black-Box Testing

Speaker: Jerry Gao Ph.D.

Computer Engineering Department
San Jose State University

email: jerry.gao@sjsu.edu
URL: http://www.engr.sjsu.edu/gaojerry

Presentation Outline

- **Software component validation**
- **Black-box testing methods for components**
  - **Random testing**
  - **Partition testing**
  - **Boundary value testing**
  - **Decision table-based testing**
  - **Mutation testing**

Software Component Validation

There are many different software validation methods for components.

They have been classified into two classes:

- **White-box validation methods (also known as program-based testing methods, or structure-based testing methods)**
  - **They refer to the systematic techniques for testers to design and generate test cases and data to achieve a certain test adequacy criteria for a component based on its component program and structure.**

- **Black-box validation methods (also known as functional testing methods)**
  - **They refer to the systematic techniques for testers to design and generate test cases and data to achieve a certain test adequacy criteria for a component based on its component specifications.**

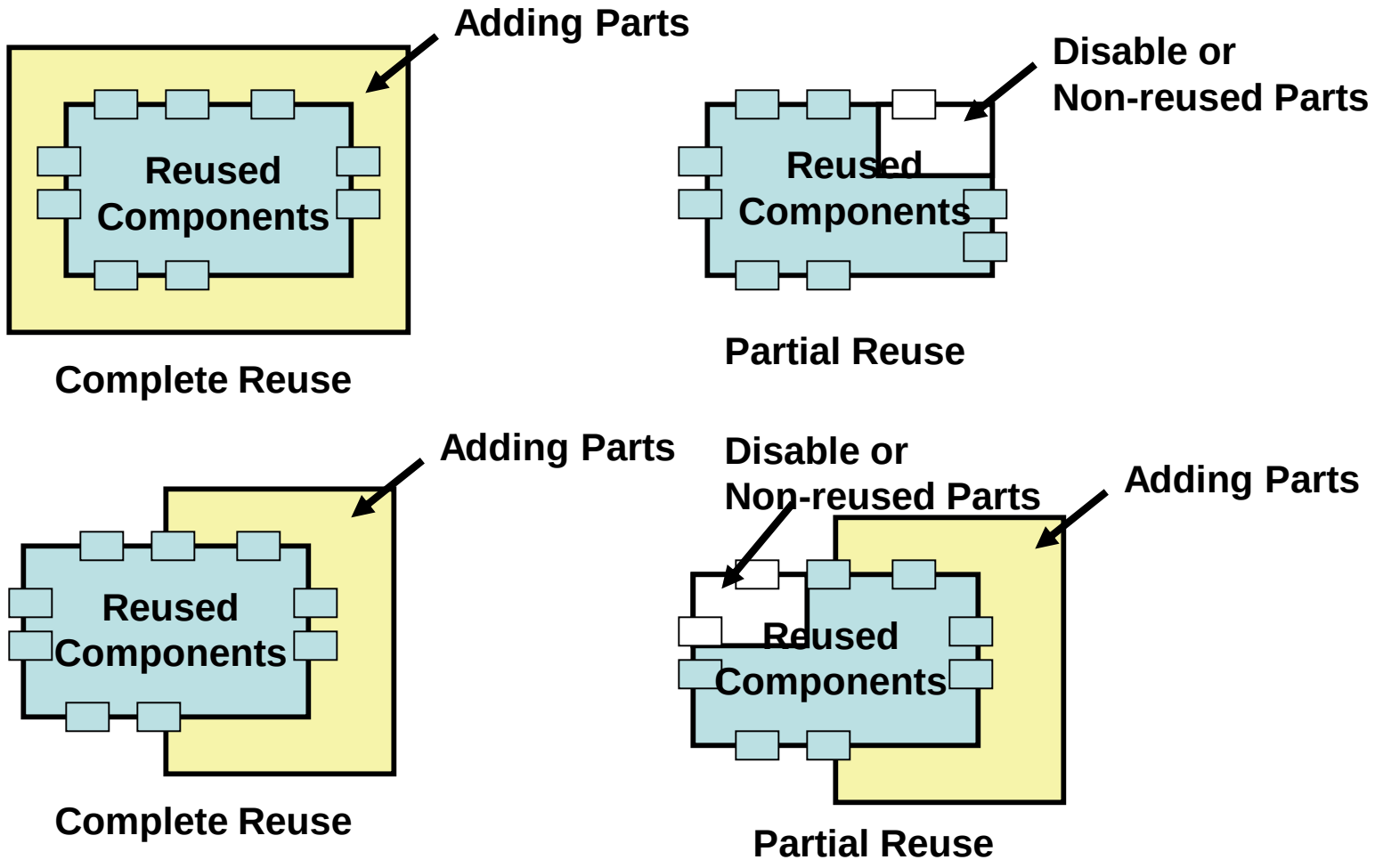
Software Component Validation

Component customization:

When a component is adopted or customized by component users,

- **Generalization customization:**
  - **A component is adopted by a user in a way to release some component requirement restrictions in component specifications.**
- **Specialization customization:**
  - **A component is adopted by a user in a way to add more requirement restrictions in component specifications.**
- **Reconstruction customization:**
  - **A component is adopted or updated by a user in a way to change input domains in component specifications.**

Different Cases for Component Validations



Black-box Testing Methods for Components

Black-box testing -> known as functional testing

Basic idea: testing a component as a black box.

- ***It focuses only on the external accessible behaviors and functions***
- ***It checks a component's outputs for selected component inputs***

In CBSE, black-box testing is very important for component users to validate and evaluate component quality.

According to IEEE Standard, black-box testing is:

> ***"Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions;"***
>
> ***"Testing conducted to evaluate the compliance of a system or component with specified functional requirements"***

The primary objective:

> ***-> to assess whether the software component does what it is supposed to do.***

Black-box Testing Methods for Components

Black-box testing methods for components can be classified into three groups:

→ ***Usage-based black-box testing techniques – focusing on user-oriented accesses***

- ***User-operation scenario testing***
- ***Random testing***
- ***Statistical testing***

→ ***Error-based black-box testing techniques – focusing on error-prone points***

- ***Equivalence partitioning testing***
- ***Category-partition testing***
- ***Boundary-value analysis***
- ***Decision table-based testing***

→ ***Fault-based black-box testing techniques – focusing on targeted fault points***

- ***Mutation testing***
- ***Fault injection method***

Black-box Testing Methods for Components

In CBSE, black-box testing can be used by component users as well as component test engineers.

For component vendors:

Their component test engineers can use all existing black-box testing methods to validate newly generated components.

For a component users:

Their engineers can apply existing black-box testing methods by pay attention to the following issues:

- ***Very limited access to quality information about COTS components, such as quality metrics and reports, and problem reports***
- ***Limited observation for component behaviors (i.e. user-level traces)***
- ***Component specification mismatch***
- ***Component customization and adoption cause test adequacy issues***
- ***Costly construction of component test beds and harness for each component***

Black-box Testing Methods for Components

Random Testing

Basic idea: randomly select input values from an input domain of a component as the inputs for test cases.

Advantages of this approach:

a) simple and Systematic b) gain certain test coverage for each input domain

Although input values can be generated and selected randomly based on input domains of a component, we need to confirm to the distribution of the input domain, or an operation profile.

i.e. ATM system: 80% withdraw, 15% deposit, %5 other transactions.

Questions in random testing for software components

→ ***Can we identify all input domains of a component?***

→ ***What are user profiles and input distribution for each input domain?***

→ ***How much of the efforts undertaken by the component providers can be reused?***

→ ***What is the targeted test coverage for input distribution in each input domain?***

→ ***How much additional testing efforts is required?***

Black-box Testing Methods for Components

Partition Testing

Basic idea: Divide the input domains of a component into N different disjoint partitions, and select one value from each input domain to create a test case.

The major problem -> Lack of systematic methods to partition input domains of a component for the given non-formal function specifications.

T. J. Ostrand and M. J. Balcer proposed a systematic method known as Category partition testing method → consisting of the following steps:

1. ***Decompose function specifications into functional units***
2. ***Identify parameters and environment conditions***
3. ***Find categories of information***
4. ***Partition each category into choices***
5. ***Write test specification for each unit***
6. ***Produce test frames***
7. ***Generate test cases***

Black-box Testing Methods for Components

Partition Testing (Test Specification Example)

Parameters:

PIN

Wrong PIN [property mismatch]

Correct PIN [property match]

Withdraw amount

Multiple of 20 [if match]

[property correct]

Less than 20 [if match]

[property wrong]

Greater than 20 but not multiple of 20

[if match]

[property wrong]

An Example of Using The Category-Partition Method

Test a command-line program that supports “find” operation as follows:

Command:

find

Syntax:

find <pattern> <file>

Function:

The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the no. of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes (“). To include a quotation mark in the pattern, two quotes in a row (“ ”) must be used.

An Example of Using The Category-Partition Method

Examples:

find john myfile

displays lines in the file myfile which contain *john.*

find “john smith” myfile

display lines in the file myfile which contains *john smith*.

find “john” ” smith” myfile

display lines in the file which contains *john” smith.*

When file is considered as a parameter, we need to consider the following:

- *no. of occurrences of the pattern in the file.*
- *no. of occurrences of the pattern in a line that contains it.*
- *maximum line length in the file*

……

An Example of Using The Category-Partition Method

Test specification for Find command:

Parameters:

- Pattern size:
  - empty
  - single character
  - many character
  - longer than any line in the file

-Quoting:
  - Pattern is quoted
  - Pattern is not quoted
  - pattern is improperly quoted

-Embedded blanks:
  - no embedded blank
  - One embedded blank
  - Several embedded blanks

An Example of Using The Category-Partition Method

Test specification for Find command:

Parameters:

- Embedded quotes:
  - no embedded quotes
  - One embedded quote
  - Several embedded quote

-File name:
  - Good file name
  - No file with this name
  - Omitted

An Example of Using The Category-Partition Method

Test specification for Find command:

Environment: (only for the pattern)

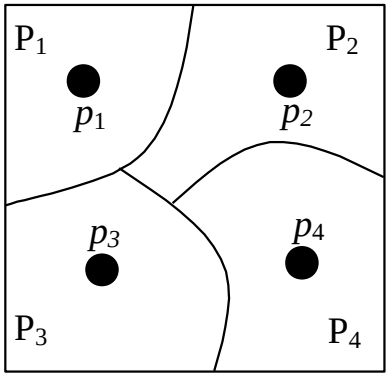
-File access environment
- File not accessible
- File can’t read
- File can’t open

-- No. of occurrences of pattern in the file.
- None
- Exactly one
- More than one

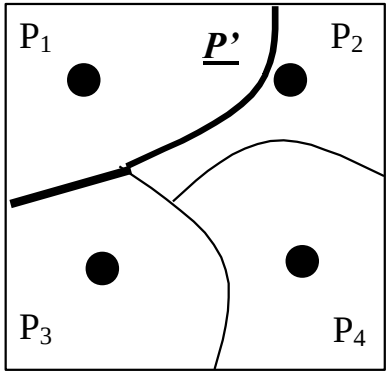
-Pattern occurrences on target line:
- one
- more than one
- None

Black-box Testing Methods for Components

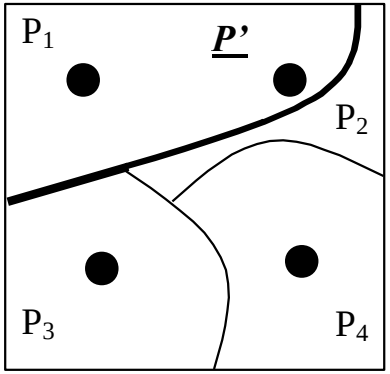
Partition Testing of Generalization Customization:



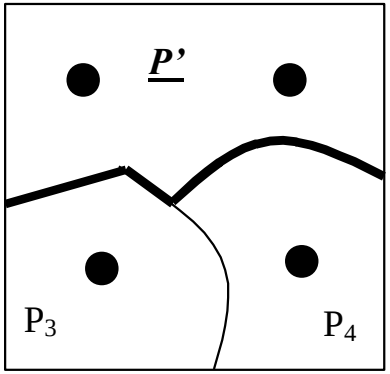
6.1 (a) Before generalization customization



6.1 (b) Case (i): After generalization customization



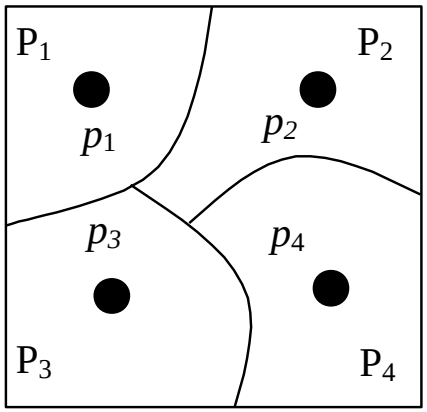
6.1 (c) Case (ii): After generalization customization



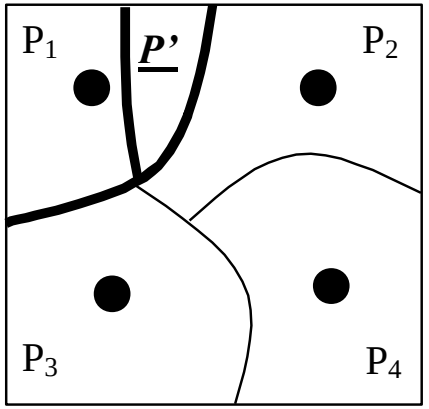
6.1 (d) Case (iii): After generalization customization

Black-box Testing Methods for Components

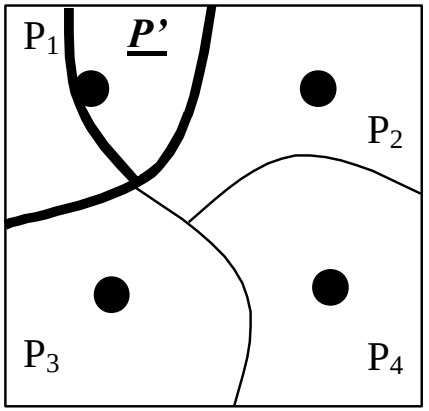
Partition Testing for Specialization Customization:



6.2 (a) Before specialization customization



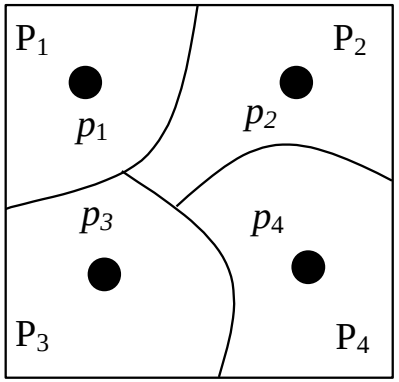
6.2 (b) Case (i): After specialization customization



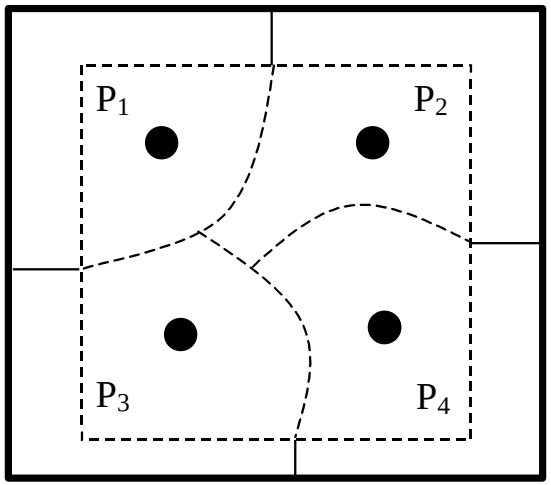
6.2 (c) Case (ii): After specialization customization

Black-box Testing Methods for Components

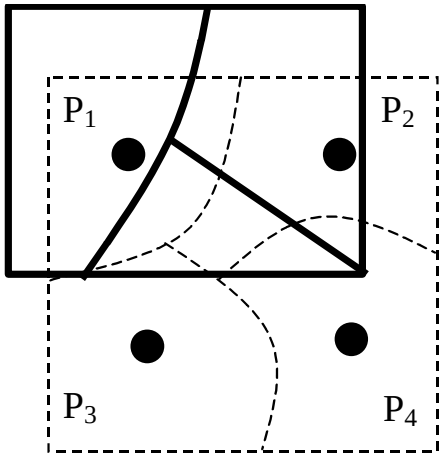
Partition Testing for Reconstruction Customization:



6.3 (a) Before reconstruction customization



6.3 (b) Special case: After reconstruction customization



6.3 (c) General case: After reconstruction customization

Black-box Testing Methods for Components

Boundary Value Testing

Basic idea:

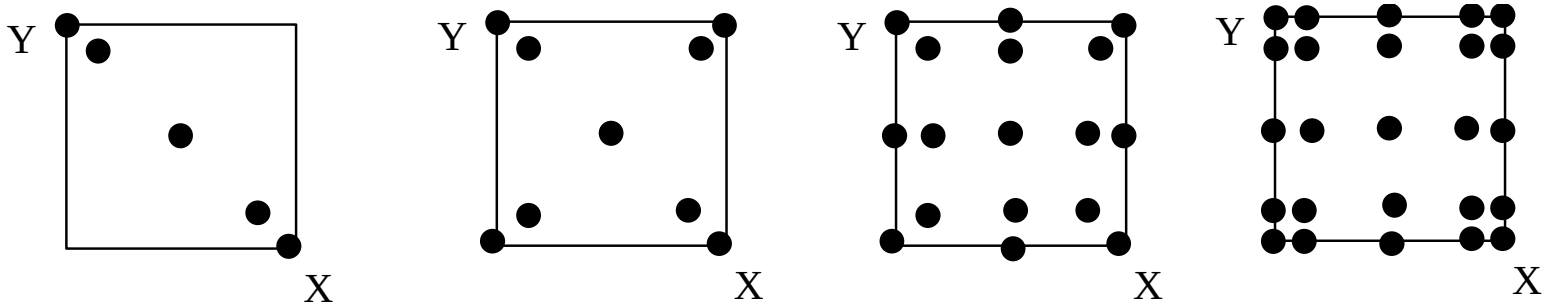
For each partition for an input domain, select boundary values for this partition to generate test cases.

Advantages of this approach:

a) easy to systematically implemented b) more like to expose errors

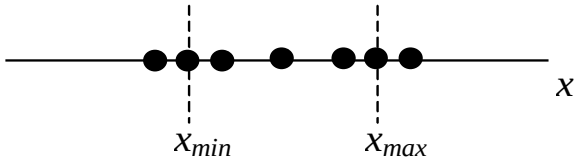
boundary values for one-dimensional space x in [X_{min}, X_{max}], y in [Y_{min}, Y_{max}]:

a. X_{min}, X_{min+1}, X_{mid}, X_{max-1}, X_{max} ***b. X_{min-1}, X_{min}, X_{min+1}, X_{mid}, X_{max}, X_{max+1}***

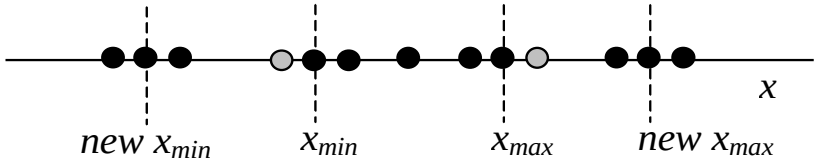


Black-box Testing Methods for Components

Boundary Value Testing:



6.5 (a) Before generalization customization

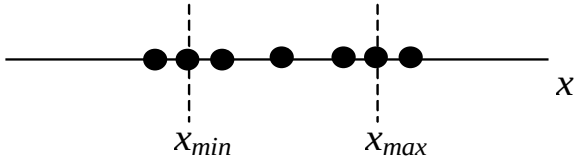


6.5 (b) After generalization customization

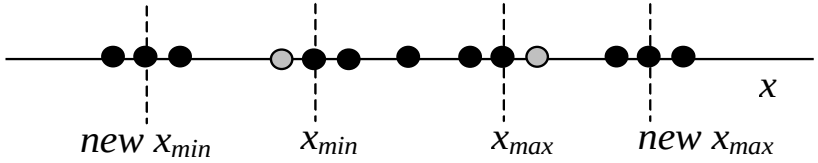
Black-box Testing Methods for Components

Boundary Value Testing – generalization customization:

→ Merge boundaries, add new boundaries and new boundary values



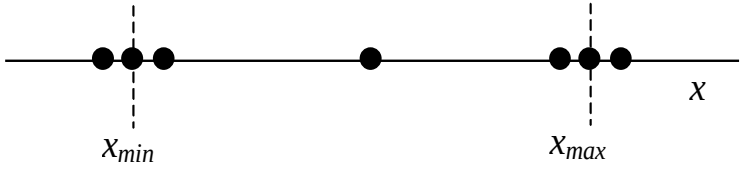
6.5 (a) Before
generalization customization



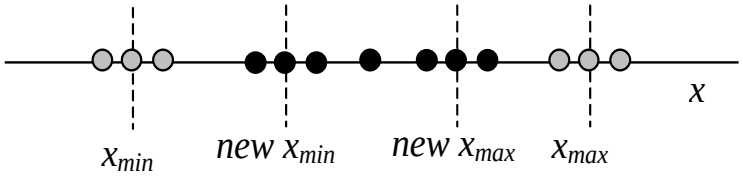
6.5 (b) After
generalization customization

Boundary Value Testing – generalization customization:

→ Add, change boundaries and new boundary values



6.6 (a) Before
specialization customization

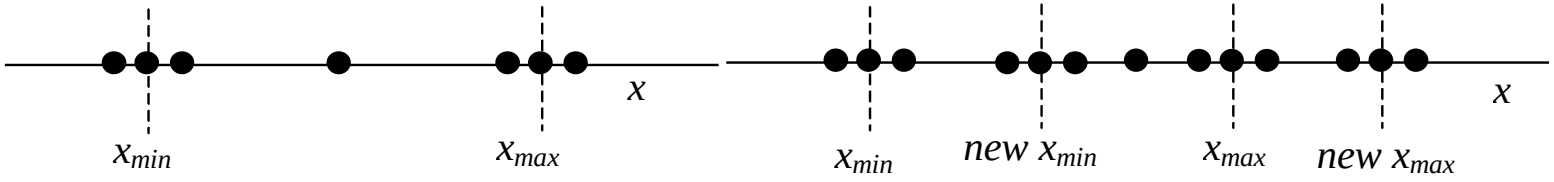


6.6 (b) After
specialization customization

Black-box Testing Methods for Components

Boundary Value Testing:

→ Introduce new boundaries and values to overlap old boundaries.



6.7 (a) Before reconstruction customization

6.7 (b) After reconstruction customization

Black-box Testing Methods for Components

Decision Table-Based Testing:

→ focuses on validating business rules, conditions, constraints and corresponding responses and actions of a software component.

Basic approach:

(a) Identify and list all possible conditions and their combination cases

(b) Identify and list all possible responding actions and outputs for each case

(c) Define test cases to cover each case

For adopted and customized software components, we need to pay the attention to:

- ***Generalization customization:***

  -> Change decision tables by removing or merging conditions and actions

- ***Specialization customization:***

  -> Change decision tables by adding new conditions and corresponding actions

- ***Reconstruction customization:***

  ->Change decision tables by adding/updating/removing conditions and related actions.

An Example

Testing a triangle analyzer:

Program specification:

Input: **3 numbers separated by commas or spaces**

Processing:

Determine if three numbers make a valid triangle; if not, print message NOT A TRIANGLE.

If it is a triangle, classify it according to the length of the sides as scalene (no sides equal), isosceles (two sides equal), or equilateral (all sides equal).

If it is a triangle, classify it according to the largest angle as acute (less than 90 degree), obtuse (greater than 90 degree), or right (exactly 90 degree).

Output: **One line listing the three numbers provided as input and the classification or the not a triangle message.**

Black-box Testing Methods for Components

Decision Table-Based Testing:

Conditions	**C1: a<b+c** **C2: a=b** **C3: b=c**	**F** **-** **-**	**T** **T** **T**	**T** **T** **F**	**T** **F** **T**	**T** **F** **F**
Actions	**Not a triangle** **Scalene** **Isosceles** **Equilateral**	**X**	**X**	**X**	**X**	**X**

Assume a≥b≥c>0. Table 6.1. Decision table based testing example

Conditions	**C1: a<b+c** **C2: a=b or b=c**	**F** **-**	**T** **T**	**T** **F**
Actions	**Not a triangle** **Scalene** **Regular**	**X**	**X**	**X**

Assume a≥b≥c>0. Table 6.2 Decision table for generalization customization

Black-box Testing Methods for Components

Decision Table-Based Testing:

Conditions	**C1: a<b+c**	**F**	**T**	**T**	**T**	**T**	**-**
	C2: a=b	**-**	**T**	**T**	**F**	**F**	**-**
	C3: b=c	**-**	**T**	**F**	**T**	**F**	**-**
	C4: $a^2=b^2+c^2$	**-**	**-**	**-**	**-**	**-**	**T**
Actions	**Not a triangle**	**X**					
	Scalene					**X**	
	Isosceles			**X**	**X**		
	Equilateral		**X**				
	Right Triangle						**X**

Assume a≥b≥c>0

Table 6.3 Partial decision table for specialization customization

Another Example – Testing a software Stack Component (class)

Assume you are given an integer stack component which is specified below. Please answer the following questions:

Stack::Create_stack(Stack S) – Create an empty queue.

Stack::Push(Stack S, int item)
– Append an item into a given stack. Please make sure the queue is not full yet when you push an item. Otherwise, an alerting message will be printed out.

Stack: Pop(Stack S, int item)
– Pop an item from a given stack. Please make sure that the stack is not empty before this operation. Otherwise, an alerting message will be printed out.

Stack: Is_Empty(Stack S)
- This function checks if the given stack is empty, and returns 1 if it is empty. Otherwise, returns 0.

(Hint: Please identify the different conditions of a Queue and related Stack's operations and results).

(a) Create a decision table to list the conditions, actions, and related results. (5%)
(b) Design and list the test cases based on the derived decision table. (5%)
Assume the maximum number of the elements in a queue is 100.

Decision Table-Based Testing Example

Decision Table-Based Testing:

Conditions	**C1: Stack = Empty** **C2: Stack = FULL** **C3: Stack existing**	**T** **F** **T**	**F** **T** **T**	**F** **F** **T**
Actions	**Create_Stack()** **Push_Item()** **Pop_Item()** **IS_Empty()**	**OUT1** **OUT3** **OUT5** **OUT7**	**OUT2** **OUT4** **OUT6** **OUT8**	**OUT2** **OUT3** **OUT6** **OUT8**

Outputs:
OUT1: “A stack is created successfully”.
OUT2: “A stack is existing”.
OUT3: “An item is pushed into a stack.”
OUT4: “The stack is full, the item can’t be pushed into the stack”.
OUT5: “The stack is empty, the pop-up action is not completed”.
OUT6: “An item is popped-up from the stack”.
OUT7: “The stack is empty”.
OUT8: “The stack is not empty”.

Black-box Testing Methods for Components

Flow Graph-Based Testing:

Objective: to check different user-oriented sequences to access component functions through its interfaces.

The basic idea: (similar to existing white-box test methods, i.e. basis-path testing)

(a) generate a flow graph for a component based on its formal specifications.

(b) use a flow graph as a test model to generate feasible flow paths.

(c) generate test cases based on executable flow paths.

(d) run the test cases to achieve defined test coverage based on the test model

Advantages: easy understand and partially systematic

Limitation:

→ ***Manual efforts in identify infeasible paths***

→ ***Highly dependent on formal component specifications***

Black-box Testing Methods for Components

Flow Graph-Based Testing: (Example given by S. H. Edward)

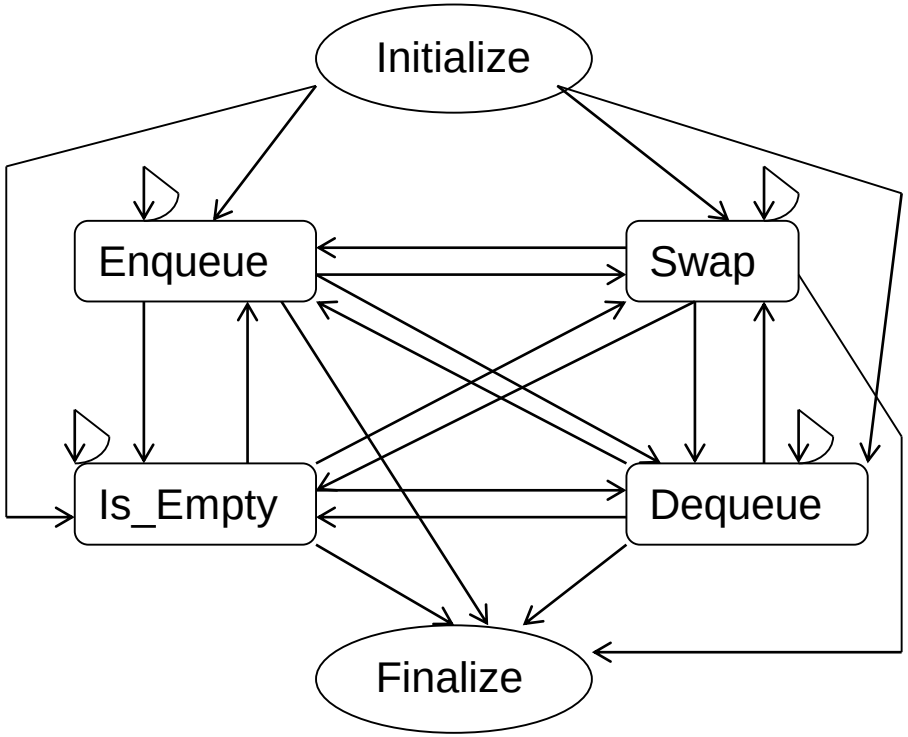


Figure 8.4. The Queue Flowgraph [6]

Black-box Testing Methods for Components

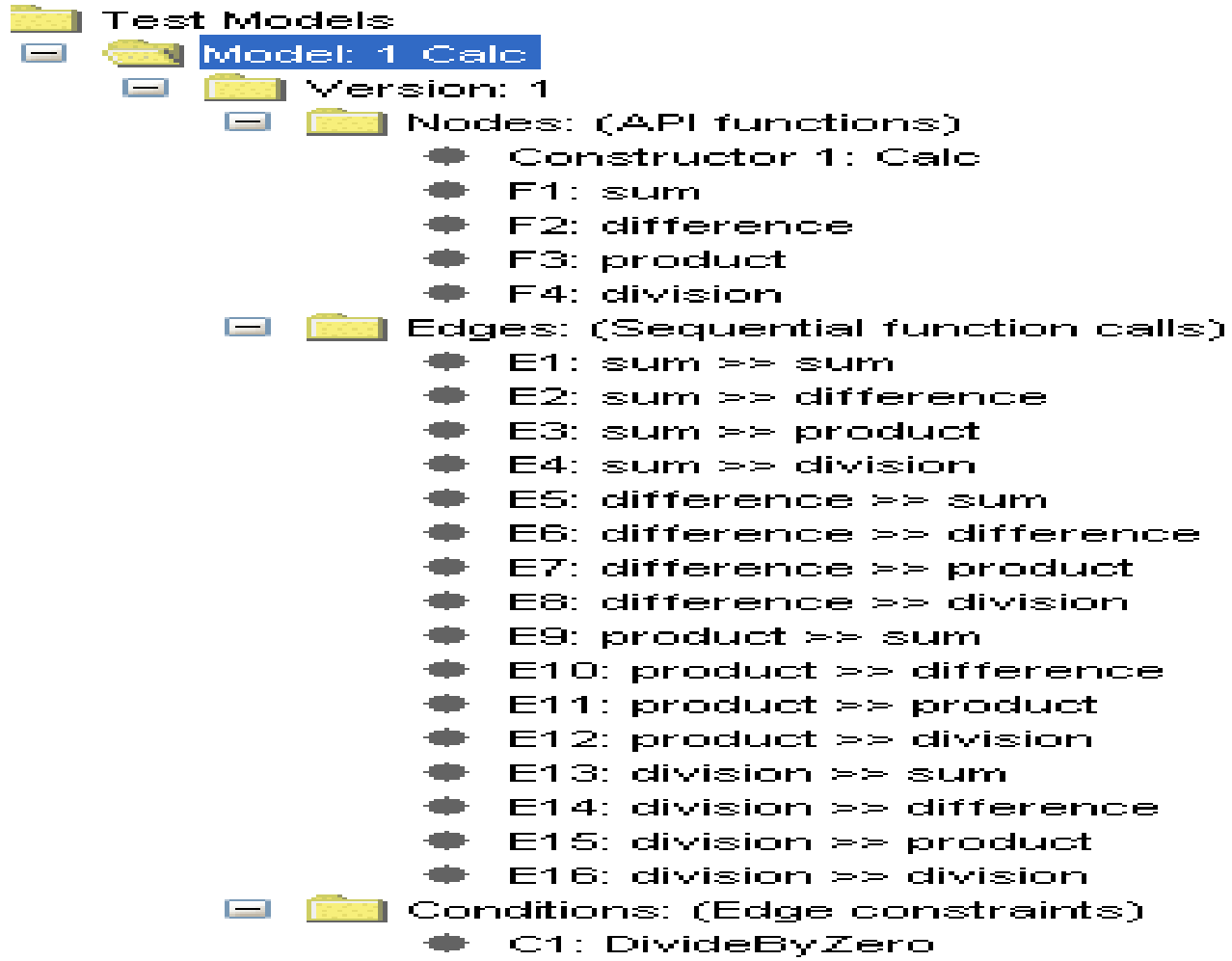
A RESOLVE queue specification:

```
concept Queue_Template
context
        global conext
            facility Standard_Boolean_Facility
        parametric context
            type Item
interface
        type Queue is modeled by string of math[Item]
            exemplar q
            initialization
                ensures     q = empty_string
        operation Enqueue (
                alters      q : Queue
                consumes    x : Item
            )
            ensures     q = #q * <#x>
        operation Dequeue (
                alters      q : Queue
                produces    x : Item
            )
            requires    q /= empty_string
            ensures             <x> * q = #q
        operation Is_Empty (
                preserves q : Queue
            ) : Boolean
                ensures     Is_Empty iff q = empty_string
end Queue_Template
```

New Black-Box Component Test Coverage Criteria

- **Component API-Based Function Access Coverage:**
  - **Focus on:**
    - **Component API-based functional accesses**
    - **Component API-based access sequences**
  - **Test model:**
    - **Component Function Access Graphs (CFAG)**
    - **Dynamic Component Function Access Graph (D-CFAG)**
  - **Test coverage:**
    - **Node coverage, all-node coverage**
    - **Transition coverage, all-transition coverage**
    - **Condition-transition coverage**
    - **Path coverage between two nodes**
    - **Minimum-path coverage between two nodes**
    - **All-nodes and all-links coverage**

An Example of CFAG Test Model



An Example of CFAG Test Model

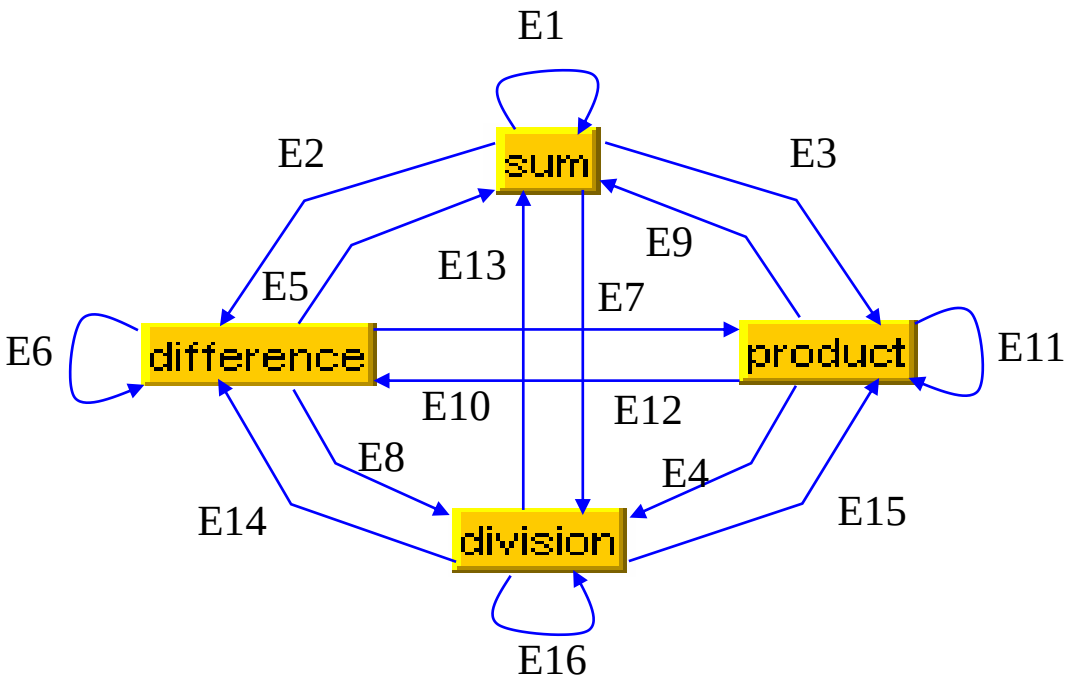


Figure 2. CFAG Test Model for a Calculator Component (Graphic Format)

Black-box Testing Methods for Components

Mutation Testing: (known as mutation analysis)

→ ***a fault-based testing technique that determine test adequacy by measuring the ability of a test suite to discriminate a component from some alternative copies (mutants).***

Basic approach: (for the same test suite)

(a) ***Introduce a single, small legal program change (syntactic level) the component***

(b) ***Create the alternative programs (mutants)***

(c) ***Execute mutants and the original program based on the same test set***

(d) ***Comparing the outputs of each test case***

(e) ***After tried all test cases in the set, if a mutant generates the same outputs as the original component, then***

→ ***the mutant = the original component.***

(f) ***Finally, compute the mutation score as follows:***

$$|dead\ mutants| \,/\, (|total\ mutants| - |equivalent\ mutants|)$$

Black-box Testing Methods for Components

Mutation Testing: (known as mutation analysis)

Basic problems:

(a) usually require source code to create mutants

(b) traditional mutation testing are very expensive due to testing of the large number of mutants

Why mutation testing is useful for component black-box testing?

→ ***We want to expose the interface errors of third-party components***

→ ***Ghosh and Mathur [23] and Edwards[24] pointed out that mutation could be an effective method to discover component interface errors .***

→ ***We can create component mutants based on component black-box interfaces.***

Testing Coverage Analysis for Software Component Validation

Raquel Espinoza
San Jose State University

COMPSAC 2005
Edinburgh, Scotland
July 26-28, 2005

Agenda

- Introduction
- Related Work
- Test Model Concept
- Coverage Criteria
- Tool Architecture
- Application Example
- Conclusion
- Recommendations

Introduction

- **Motivation:** support widely used method of incorporating reusable software components to develop larger software systems to reduce development cost, effort and time
- **Problem:** How to adequately test these large software systems prior to release
  - If individual reused software components are not adequately tested, it may be difficult and time consuming to isolate the source of a problem
- **Solution:** Begin testing at the component level to weed out component quality problems and address reuse context issues prior to incorporation
- **Proposed Method:** A model-based coverage analysis method to address adequate testing of a components API in the validation for its reuse

Related Work

- S. H. Edwards define a black-box test model, known as flow graphs, derived from a component's specification and applied analogues to traditional graph coverage techniques to develop a test set generation approach.
- D. Hoffman and P. Strooper define a state-based test model, known as a testgraph, to model a class-under-test (CUT) as a state machine and applies graph traversal techniques for test input generation and develops driver and oracle classes to support test execution.
- S. Beydeda and V. Gruhn define a test model derived from a class specification and implementation and present it as a control flow graph to support structural testing techniques, such as coverage criteria-based techniques, for use in test case generation.
- D. S. Rosenblum defines two formal test models that address adequate unit testing and integration testing of components. These models are defined based on the subdomain-based test adequacy criteria defined by Frankl and Weyuker.

Proposed Method

- A model-based coverage analysis method for a component API to address adequate testing issue in validation for reuse
  - Define a test model that represents a component's API interface
  - Identify coverage criterion for a test model
  - Dynamically analyze and monitor test coverage by evaluating test scenarios to determine percent of coverage provided

Academic Contribution

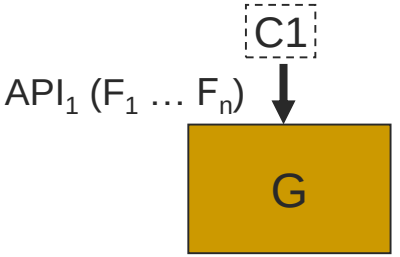
- To establish a dynamic test coverage analysis method that provides identification of higher quality test sets that can offer adequate test coverage at the component level
- To offer a more effective testing method for a components API

Test Model Concept

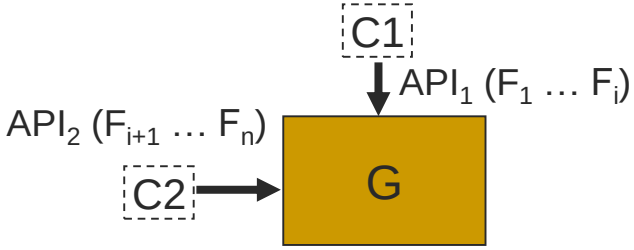
- A test model represents a component's API interface and semantics that define constraints and usage of the interface
  - Each accessible interface method is defined as a "*node*"
  - Each possible interface method call between a pair of nodes is defined as an edge or "*link*"
  - A link may be "*conditional*" when an accessible element or state of the component represents an operation constraint of the interface method call

Component API Interface

- **Single API**
  - API has a complete set of accessible methods
  - API defines usage by another system component



- **Multiple APIs**
  - Each API has subset of accessible methods
  - Each API defines usage by different system components



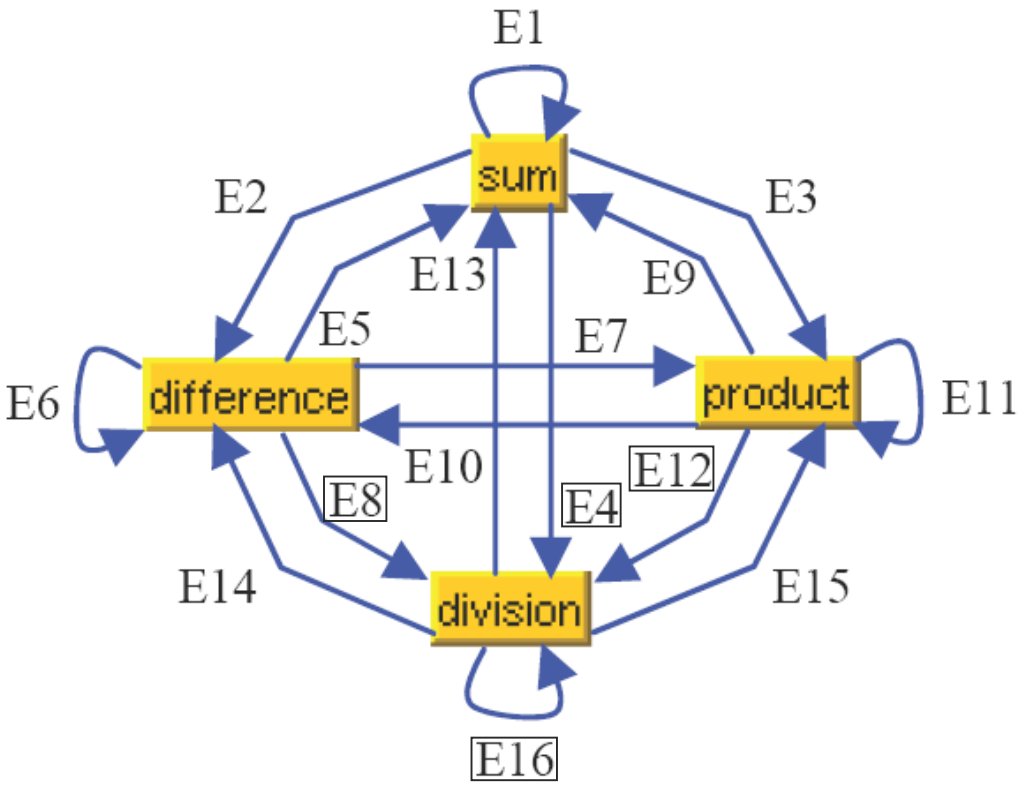
Test Model Notation

- Analytical expression: G = (F,E)
  - G: test model for a component
  - F: set of all nodes
  - E: set of all links
- F_i is accessible interface method of G
- $E_n = (F_i, F_j)$ describes the method access sequence of F_i then F_j

Test Model Representation

- Types
  - Component Functional Access Graph (**CFAG**): visual representation of the test model structure
  - Dynamic Component Functional Access Graph (**D-CFAG**): visual representation of the execution sequence of a test scenario for a given CFAG
- Purpose
  - To assist engineers in defining various test criteria
  - To facilitate automatic test generation
  - To facilitate test coverage analysis

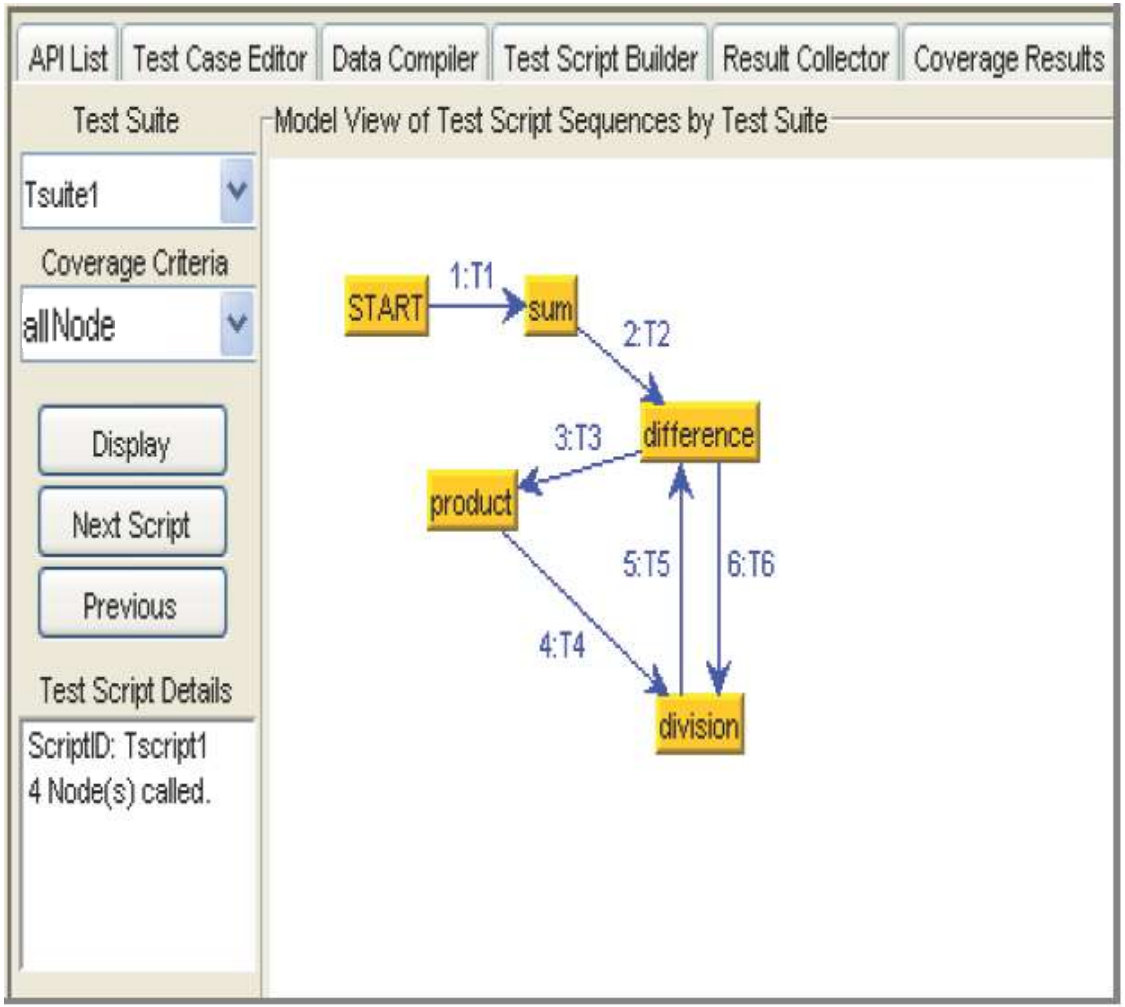
CFAG Example:
Simple Calculator Component



- Single API
- **Nodes:** sum, product, difference, division
- **Links:** E1 thru E16
- **Conditional links:** E4, E8, E12 and E16 where the condition addresses division by zero
- **Total links:** n^2 where n = number of nodes

D-CFAG Example:
Simple Calculator Component

- One possible test scenario with 6 test cases
- All 4 nodes exercised
- Only 5 out of 16 links exercised



Coverage Criteria

- Metric used to assess the effectiveness of a set of tests to exercise the functionality offered by a software component for the purpose of exposing defects
- Coverage criteria target specific characteristics of the software component's structural or behavioral properties
- Coverage criteria considered:
  - Node coverage criteria for each accessible method in an API interface
  - Link coverage criteria for each link between two nodes
  - Conditional link coverage criteria
  - Path coverage criteria for API access sequences between the nodes

Coverage Criterion

- **Node:** achieved for node F_i in G if and only if its adequate test set TF_i has been exercised
- **All-node:** achieved if and only if every node F_i in G has achieved its node coverage criterion
- **Link:** achieved for link $E_n = (F_i,F_j)$ in G if and only if F_i has been exercised at least once using the test set TF_i followed by exercising F_j using the test set TF_j
- **All-link:** achieved if and only if every node E_n in G has achieved its link coverage criterion
- **Condition-link:** achieved for conditional link E_i in G if and only if E_i is exercised with two test cases such that both TRUE and FALSE conditions are tested
- **All-condition link:** achieved if and only if every conditional node E_i in G has achieved its condition-link criterion
- **Path:** achieved for P_k in G where $P_k = (E_i, E_j \ldots E_n)$ if and only if P_k is exercised at least once in a sequence $E_i, E_j \ldots E_n$ by a test script in the test set T for G.
- **Minimum-set path:** achieved if and only if there exists a path set P (for E_i to E_n) which covers all nodes and links reachable from E_i to E_n and is traversed by test scripts in test set T.

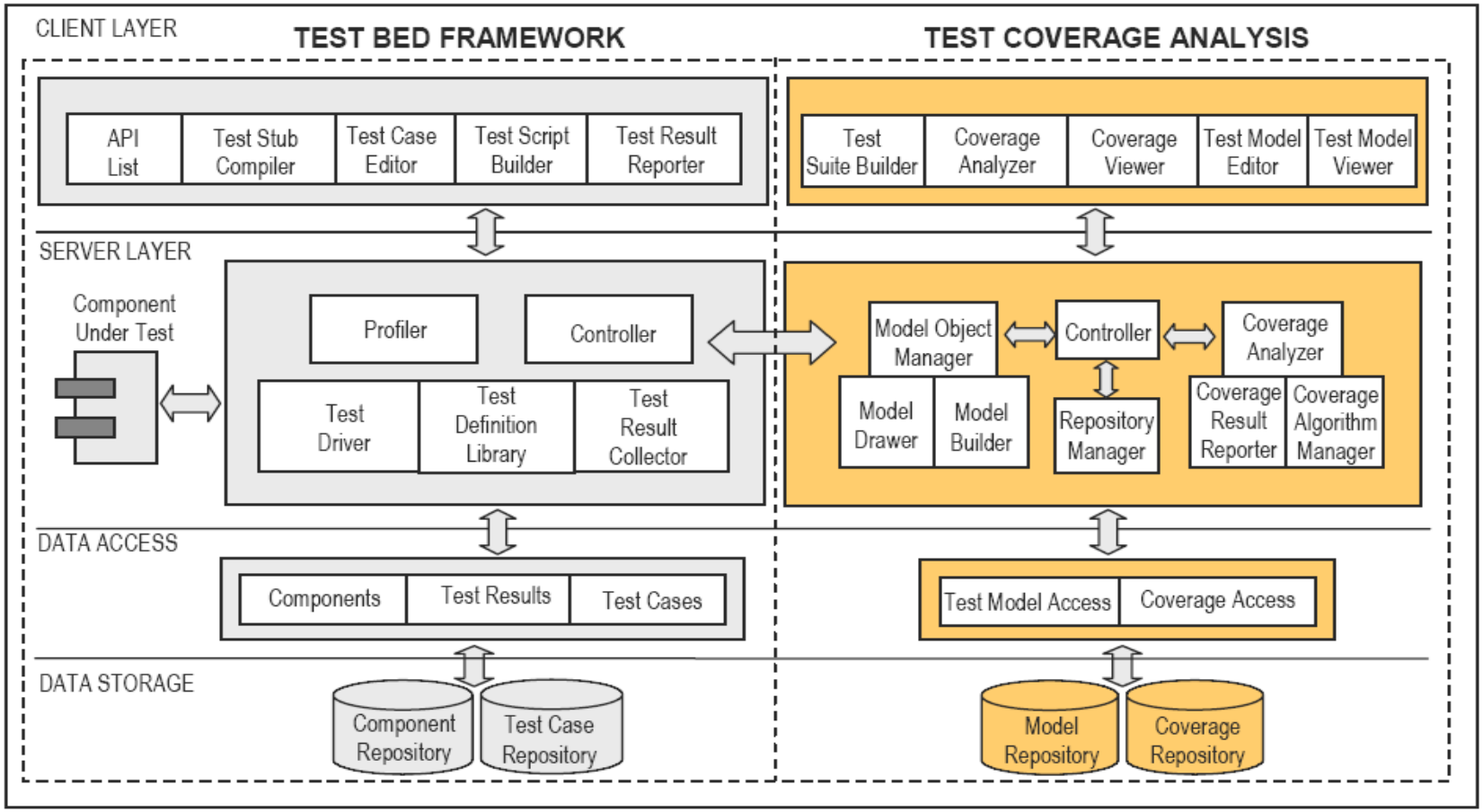
Applying Test Coverage to API

- Component C has a single API interface and black-box test set T with N test scenarios
  - **CFAG** for C is **G = (F, E)**
  - **i-th D-CFAG** for C that represents the i-th test scenario in T from 1 to N is **G_i = (F[i], E[i])**

- Determine set of nodes and links achieving desired coverage using union operation
  - Covered-Node-Set = F[1] U F[2] U … U F[N]
  - Covered-Link-Set = E[1] U E[2] U … U E[N]
  - Uncovered-Node-Set = F – Covered-Node-Set
  - Uncovered-Link-Set = E – Covered-Link-Set

Tool Architecture

- Incorporated concepts into an automated testing application to provide test model generation and coverage analysis



Tool Functionality

- Test coverage analysis
  - Automatic generation of test models (CFAG)
  - Editing of test models
  - Creating test suites (sets of test scenarios)
  - Setup of the coverage analyzer for selected criterion
  - Executing of the coverage analysis
  - Viewing test scenario sequences (D-CFAG)
  - Viewing of coverage results
  - Storing test models and coverage results
- Test-bed framework
  - Loading profiles of component's API
  - Creating test case and test scripts
  - Executing test scripts on the loaded components
  - Viewing test result

Application Example:
BTree Component

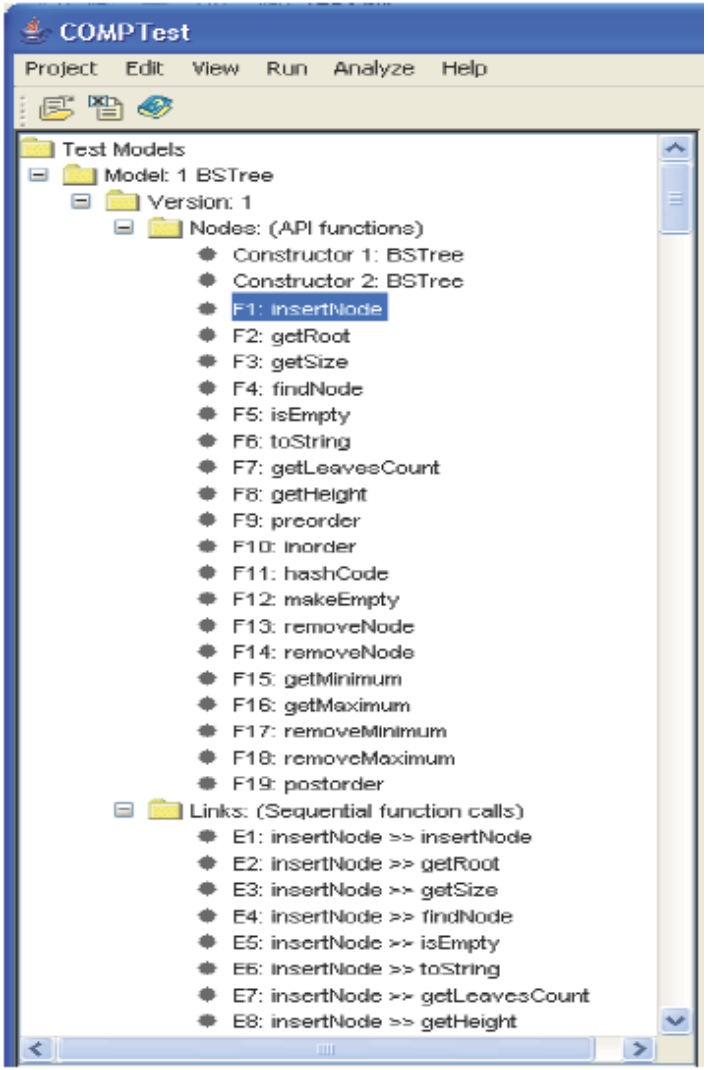
- Single API
- Component includes two classes
  - BSTree: API interface
  - BSTNode: object class

- Test set T containing three test scenarios T_1, T_2 & T_3 generated to exercise in different ways the inserting elements, removing elements and verifying BTree structure after insertion/removal
  - T_1 execute 18 method calls
  - T_2 & T_3 execute 13 method calls

- All test scenarios were analyzed for both node and link coverage

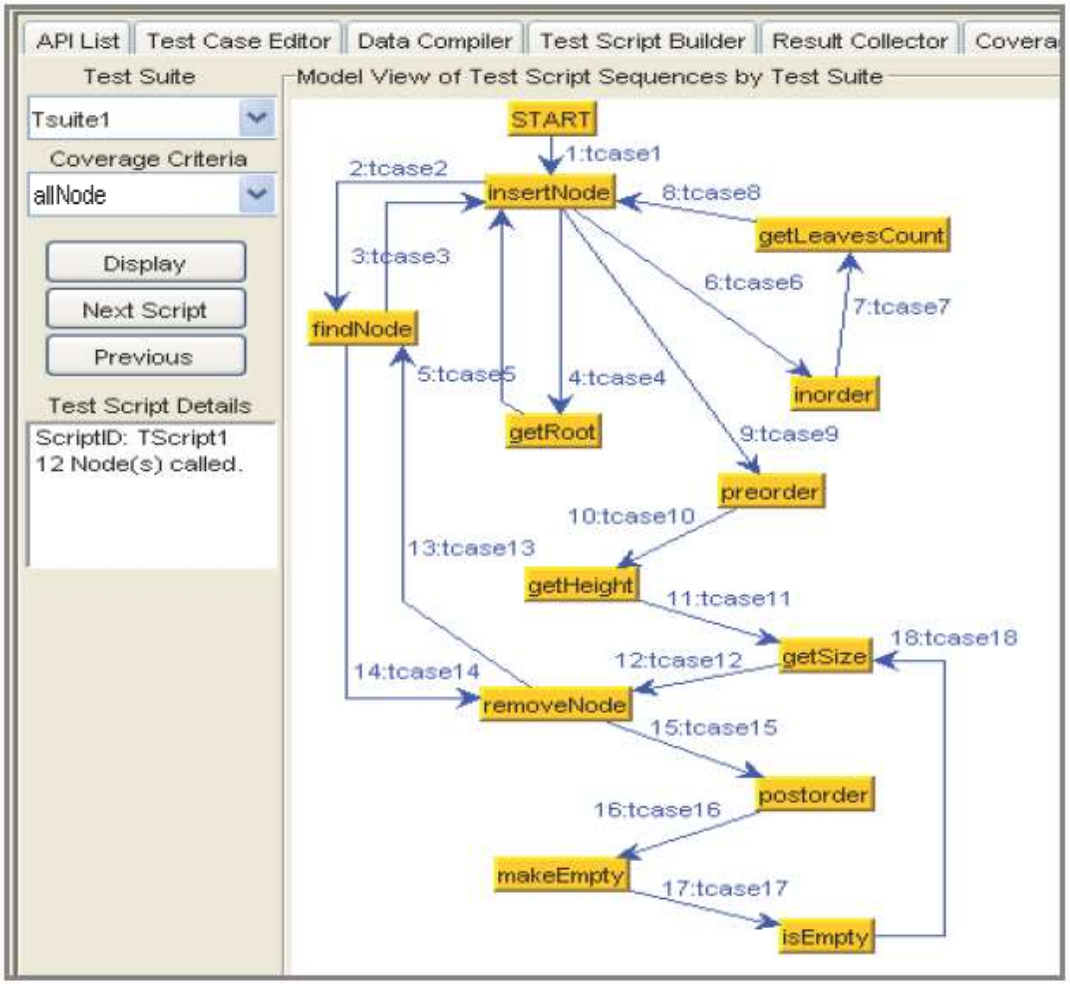
CFAG Representation

- BTree interface has 19 accessible methods
  - G = (F,E)
  - F = F_1 … F_{19}
  - E = E_1 … E_{361}
- A tree structure is used to display the CFAG
- Selection of a node or link will display more information such as link conditions



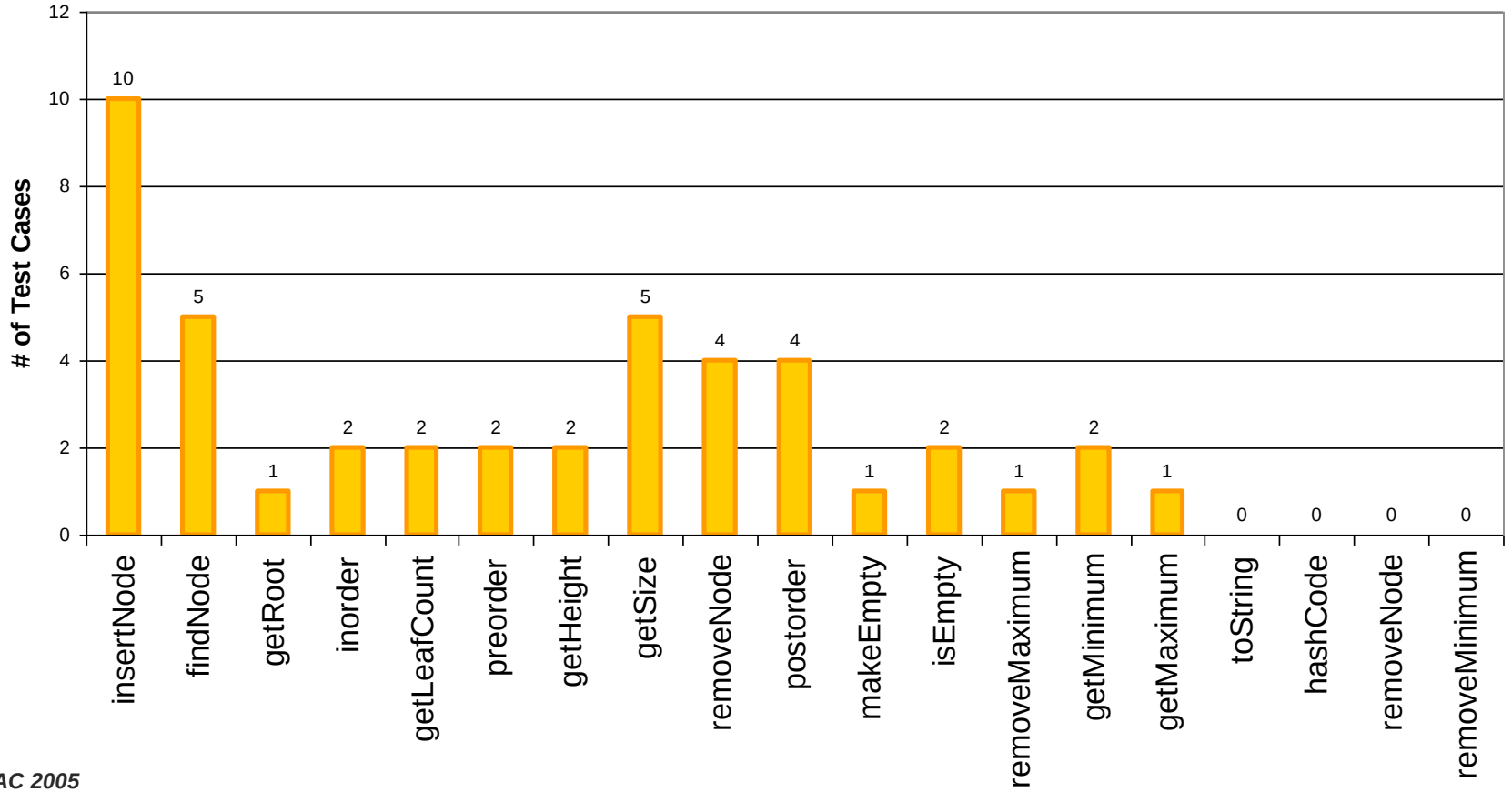
D-CFAG Representation

- $G_{T1} = (F[T_1], E[T_1])$ (SHOWN)
  - 18 calls to 12 nodes
  - Node coverage 55%
  - Link coverage 4.5%
- $G_{T2} = (F[T_2], E[T_2])$
  - 13 calls to 8 nodes
  - Node coverage 45%
  - Link coverage 2.9%
- $G_{T3} = (F[T_3], E[T_3])$
  - 13 calls to 11 nodes
  - Node coverage 55%
  - Link coverage 3.2%



Coverage Analysis Result: Node Coverage Criterion

- Covered-Node-Set = $F[T_1] \cup F[T_2] \cup F[T_3]$
- 3 test scenarios provided a total node coverage of **78.9%**



Coverage Analysis Result: Link Coverage Criterion

- Covered-Link-Set = $E[T_1] \cup E[T_2] \cup E[T_3]$
- 3 test scenarios provided total link coverage of **9.4%**

API Methods	insertNode	findNode	getRoot	inorder	getLeafCount	preorder	getHeight	getSize	removeNode	postorder	makeEmpty	isEmpty	removeMaximum	removeMinimum	getMinimum	getMaximum	toString	hashCode	removeNode
insertNode		1	1		1	1										1			
findNode	3												1						
getRoot	1																		
inorder	2																		
getLeafCount				1			1												
preorder	1					1													
getHeight		1				1													
getSize	1				1		1					2							
removeNode		1		1				1		1									
postorder	1							1	1						1				
makeEmpty										1									
isEmpty									1		1								
emoveMaximum										1									
emoveMinimum																			
getMinimum	1							1											
getMaximum															1				
toString																			
hashCode																			
removeNode																			

Conclusion

- A basic framework is proposed for generating test models (CFAG and D-CFAG) based on a component's API interface and developing a set of coverage criteria for the models
- Challenges for this method:
  - Developing a test model that sufficiently represents an component API
  - Displaying the test model and coverage results in a comprehensive manner
  - Defining adequate coverage criterion for the test model
  - Defining a test set that provides adequate test coverage
- This method provides revealing coverage metrics by facilitating the identification of higher quality test sets that can offer adequate test coverage
- It is a first step towards defining a more effective component-level unit testing method for a component's API

Recommendations and Future Work

- Develop coverage criteria that identify high risk areas that should be tested
- Develop test models and test coverage methods to address adequate testing for component reuse at the component integration and system level
- Future research efforts will be dedicated to:
  - Develop a systematic solution for a minimum path-set
  - Apply current research to components with different types of interfaces and components with multiple APIs
  - Extent the test models to represent component interaction patterns and develop related coverage criteria

Component Testability

San José State University

San José State
UNIVERSITY

By Jerry Zeyu Gao, Ph.D.
San Jose State University
Email: jerry.gao@sjsu.edu
URL:/www.engr.sjsu.edu/gaojerry

Presentation Outline

- **Basic concepts of software testability**
- **Understanding component testability**
  - **Component understandability**
  - **Component observability**
  - **Component traceability**
  - **Component controllability**
  - **Test support capability**
- **Design for component testability**
  - **Understanding different approaches to increasing component testability**
  - **Building BIT components**
  - **Building testable components**
- **Verification and evaluation of component testability**
- **Software testability measurement**

Basic Concepts of Software Testability

Testability is a very important quality indicator of software and its components since its measurement leads to the prospect of facilitating and improving a software test process.

"The degree to which a system or component facilitate the establishment of test criteria and the performance of tests to determine whether those criteria have been met; the degree to which a requirement is stated in terms that permit the establishment of test criteria and performance of tests to determine whether those criteria have been met." (by IEEE Standard)

Software testability depends on our answer to the following questions:

- → **Do we construct a system and its components in a way that facilitates the establishment of test criteria and performance of tests based on the criteria?**

- → **Do we provide component and system requirements that are clear enough to allow testers to define clear and reachable test criteria and perform tests to see whether they have been met?**

Basic Concepts of Software Testability

Poor testability of components and programs indicate:

- → ***the poor quality of software and components***
- → ***the ineffective test process***

Although verifying and measuring software testability after implementation is useful for quality control, it is little too late to increase and enhance the testability of software and its components.

In a practice view, we need to focus on the two areas:

- → ***How to increase software testability by constructing highly testable components and systems?***

- → ***How to analyze, control, and measure the testability of software components in all phases of a software development process?***

Different Tasks and Activities Relating Testability

Requirements Analysis:

- → ***Clearly define testable and measurable requirements***
- → ***Provide well-defined facilitating requirements for software testing***
- → ***Review and evaluate system/component requirements to make sure they are testable and measurable***

Software Design:

- → ***Conduct design for testability of software and components, i.e. come out architecture model and interfaces increasing testability.***
- → ***Define design patterns, standards, and framework for testable components***
- → ***Review and evaluate software design concerning software testability***

Implementation:

- → ***Implement testable components and built-in tests***
- → ***Generate software testing facilities and reusable framework***
- → ***Review program/component code based on well-defined testability standards***

Different Tasks and Activities Relating Testability

Testing:

- → ***Define achievable component test criteria and high quality tests/scripts***
- → ***Develop, set-up, and use test beds and facilities for components***
- → ***Perform component tests and monitor coverage based on defined criteria***
- → ***Verify, evaluate, and measure component testability***

Maintenance:

- → ***Update and review test criteria and tests based on component changes***
- → ***Maintain testable components and built-in tests***
- → ***Maintain component test framework and test beds***
- → ***Evaluate, verify, and measure the testability of components***

Understanding Component Testability

What is software component testability?

- R. S. Freedman defined component testability in a function domain by considering two factors:

(a) Component Observability and (b) Component Controllability

- R. V. Binder discussed testability of object-oriented programs by considering six factors:

(A) Representation, (B) Implementation, (C) Built-in Test

(D) Test Suite, (E) Test Support Environment, and (F) Process Capability

Component testability is two-fold:

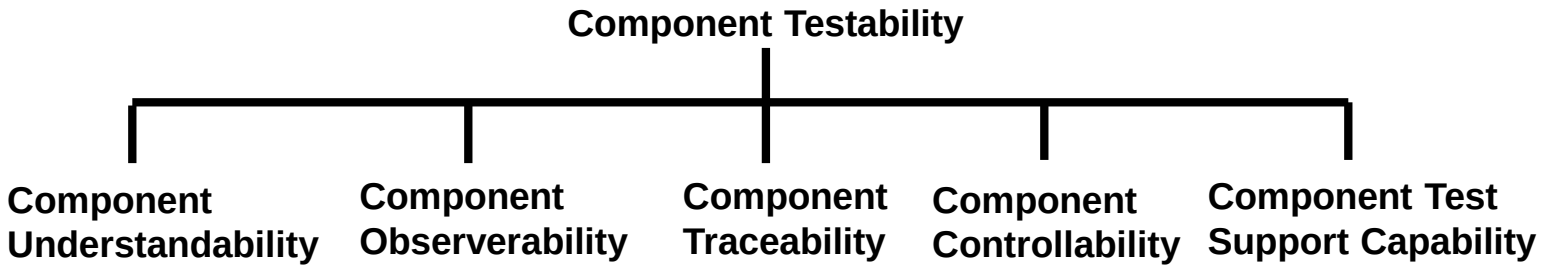
- It refers to the degree to which a component is constructed to facilitate the establishment of component test criteria and the performance of tests to determine whether those criteria have been met.

- It refers to the degree to which testable and measurable component requirements are clearly given to allow the establishment of test criteria and performance of tests.

Understanding Component Testability

Software component testability depends on the following five factors:

- → **Component understandability**
- → **Component observability**
- → **Component traceability**
- → **Component controllability**
- → **Component testing support capability**



Studying component testability focuses on two aspects:

1. **Studying how to construct testable components, including component development methods, guidelines, principles, and standards**
2. **Studying how to verify and measure component testability based on established test criteria**

Component Understandability

Component understandability depends on the following two factors:

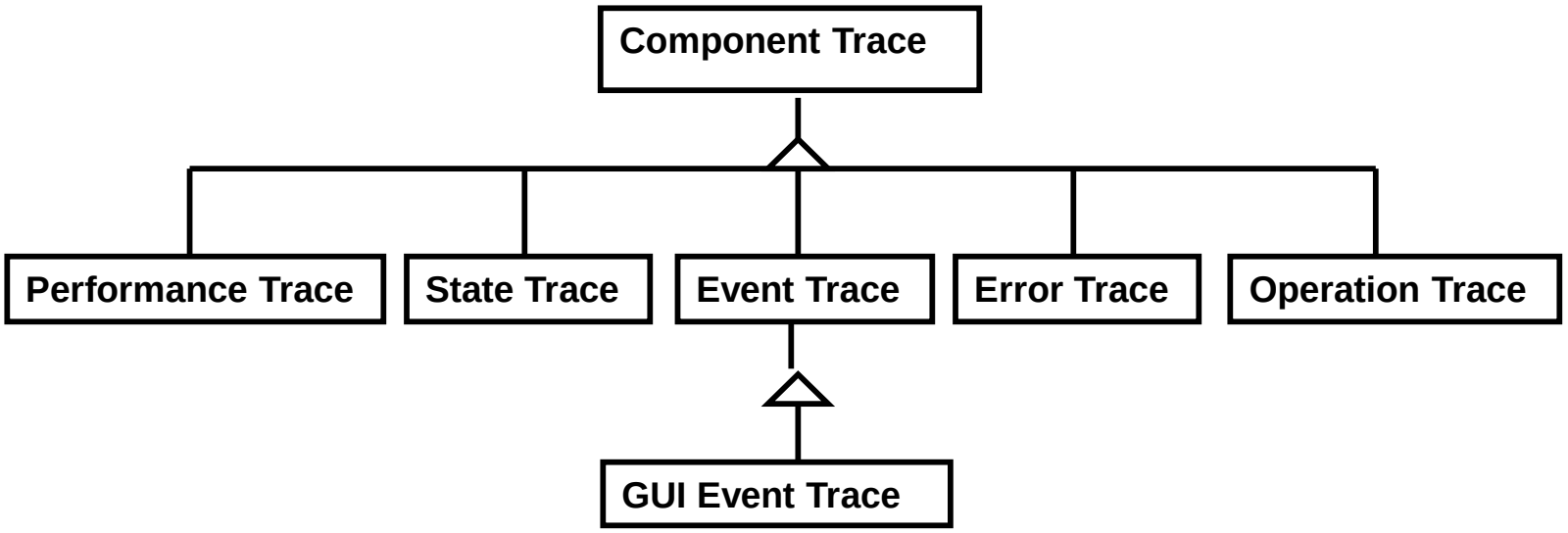
- → **Availability of component artifacts for component users, such as component function specifications, interfaces, programs, testing doc.**
- → **Availability of component artifacts for component developers, such as Component user manual and reference documents.**
- → **Understandability of component artifacts.**

Component observability indicates how easy it is to observe a program based on its operation behaviors, input parameter values, and actual outputs for a test case.

Component Traceability

Component traceability depends on the following five factors:

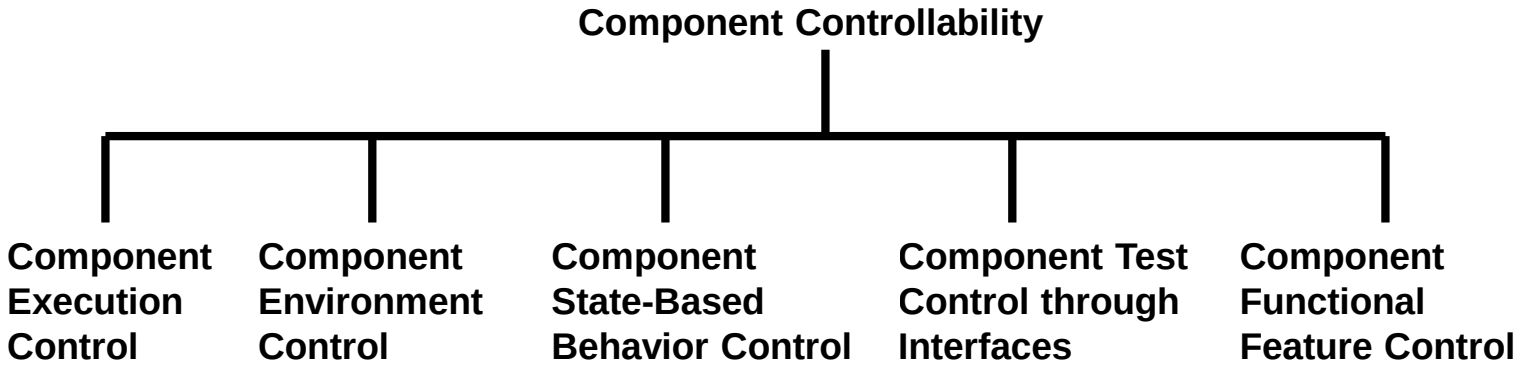
- → **Component error traceability**
- → **Component state behavior traceability**
- → **Component event traceability**
- → **Component function/operation traceability**
- → **Component performance traceability**



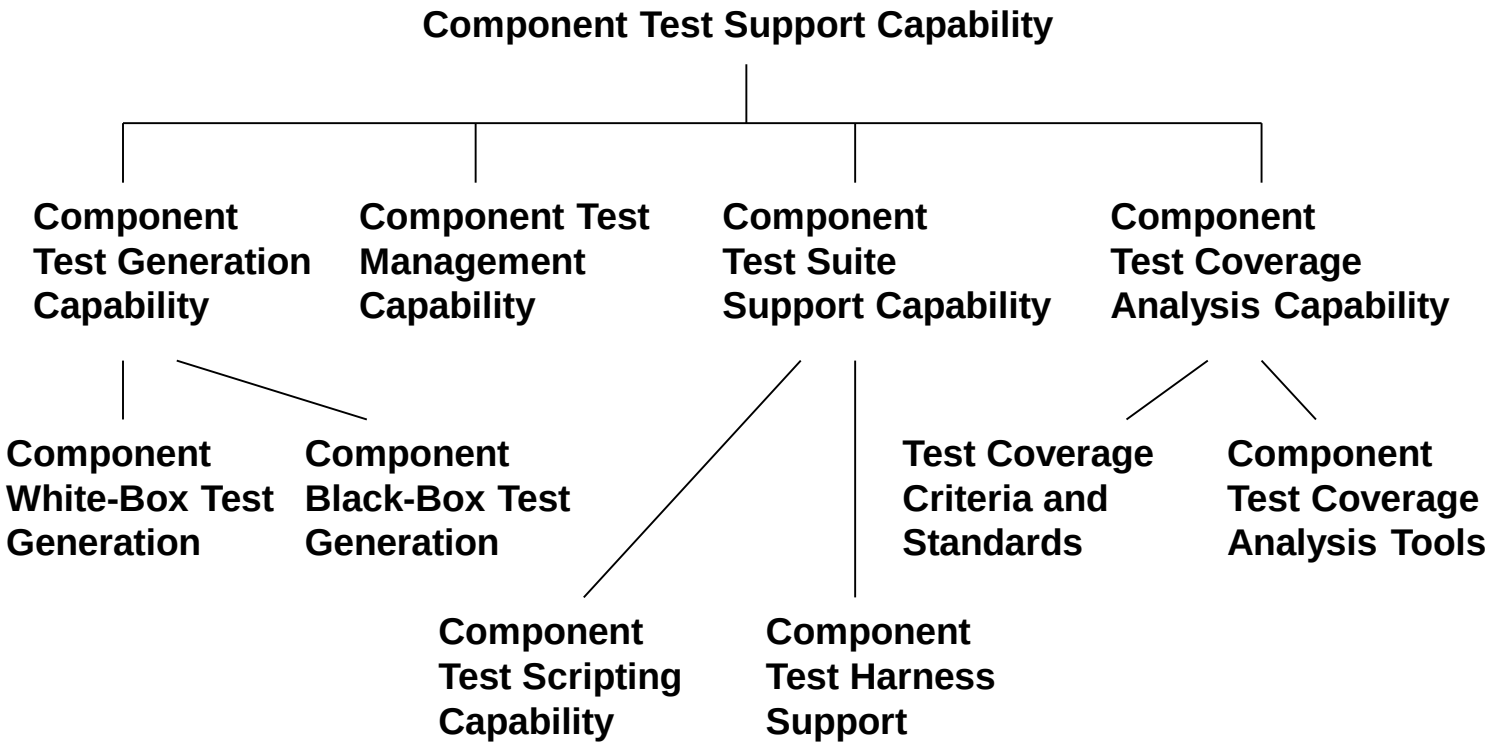
Component Controllability

Component controllability depends on the following five factors:

- → **Component execution control**
- → **Component environment control**
- → **Component state-based behavior control**
- → **Component test control through interfaces**
- → **Component functional feature control**



Component Test Support Capability



Component Testability Issues and Challenges

Component testability issues in CBSE:

- → **How to construct components with high testability? (in other words, how to create testable software components?)**
- → **How to increase component testability in a component reuse process?**
- → **How to check component testability during a component development process?**
- → **How to measure component testability in a component development process?**

Challenges in studying component testability:

- → **Creating component testability models**
- → **Finding systematic methods to create testable components**
- → **Developing systematic methods to verify component testability**
- → **Defining measurement methods and metrics for component testability**

Design for Component Testability

Design for component testability refers to all engineering activities to enhance component testability for software components in a component development process.

Challenges in building testable components:

- → **How to specify testability requirements for components?**
- → **How to construct components to achieve high testability? (including construction approaches, component architecture, test interface, ….)**
- → **How to support test automation for testable components?**
- → **How to verify generated component testability in a systematic solution?**
- → **How to measure and analyze the testability of components during a component development process in a systematic approach?**

Three Common Approaches

Three common approaches to increase component testability:

- ***Method #1: Framework-based testing facility***

  → Creating well-defined framework (such as a class library) is developed to allow engineers to add program test-support code into components according to the provided application interface of a component test framework.

- ***Method #2: Build-in tests***

  → Adding test-support code and built-in tests inside a software component as its parts to make it testable.

- ***Method #3: Systematic component wrapping for testing***

  → Using a systematic way to convert a software component into a testable component by wrapping it with the program code that facilitates software testing.

Built-in Test Components

Definition:

According to Y. Wang et al, a built-in test component is a special type of software component in which special member functions are included as its source code for enhancing software testability and maintainability.

Major features:

- **Built-in test components are able to operate in two modes:**
  - **a) normal mode – a component behaviors as its specified functions.**
  - **b) maintenance mode – its internal built-in tests can be activated by interacting a tester (or user).**
- **Built-in tests as a part of a component. (see an example)**

Major limits:

- **Only limited tests can be built-in tests due to component complexity**
- **It is costly to change and maintain built-in tests during a component development process.**

Comparison of Three Approaches

Different Perspectives	Framework-Based Testing Facility	Built-in Tests	Systematic Component Wrapping for Testing
Programming Overhead	Low	High	Very Low
Testing Code Separated from Source Code	No	No	Yes
Software Tests inside Components	No	Yes	No
Test Change Impact on Components	No	High	No
Software Change Impact on Component Testing Interfaces	No	Yes	No
Component Complexity	Low	Very High	High
Usage Flexibility	High	Low	Low
Applicable Components	In-house components and newly developed components	In-house components and newly developed component	In-house components and COTS as well as newly constructed components

What Is A Testable Component?

"A testable bean is a testable software component that is not only deployable and executable, but is also testable with the support of standardized components test facilities." (by Jerry Zeyu Gao et al.)

The basic requirements of a testable bean:

***Requirement #1: A testable bean should be deployable and* executable.**

A JavaBean is a typical example.

***Requirement #2:* A testable bean must be traceable by supporting basic component tracking capability that enables a user to monitor and track its behaviors.**

***Requirement #3:* A testable bean must provide a consistent, well-defined, and built-in interface, called component test interface, to support external interactions for software testing.**

***Requirement #4:* A testable bean must include built-in program code to facilitate component testing by interacting with the two provided test interfaces to select tests, set up and run tests, and check test results.**

Why Do We Need Testable Components?

The major goal of introducing testable components is to find a new way to develop software components which are easily to be observed, traced, tested, deployed, and executed.

The major advantages of testable components:

- ***Increasing component testability by enhancing component understandability, observability, controllability, and test support capability.***
- ***Standardizing component test interfaces and interaction protocols between components and test management systems and test suite environments.***
- ***Reducing the effort of setting up component test beds by providing a generic plug-in-and-test environment to support component testing and evaluation.***
- ***Providing the basic support for a systematic approach to automating the derivation of component test drivers and stubs.***

Principles of Building Testable Components

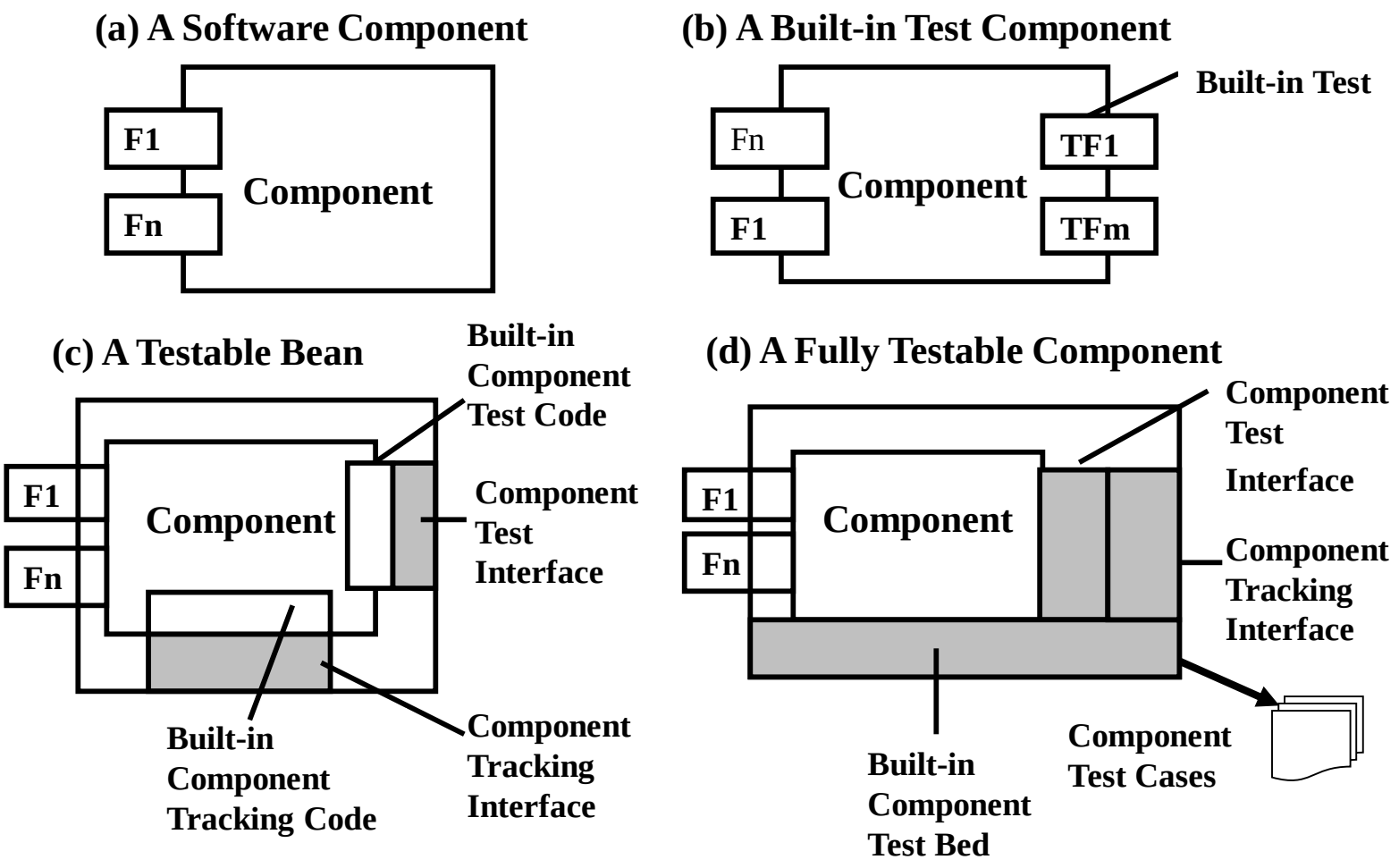
The essential needs in constructing testable components are:

- ***Well-defined component models concerning test support***
- ***Consistent test interfaces between components and external test tools and facilities***
- ***Effective ways and mechanisms to construct testable components***

The basic principles of building testable components:

- ***It is essential to minimize the development efforts and program overheads when we increase component testability by providing systematic mechanisms and reusable facilities.***
- ***It is important to standardize component test interfaces for testable beans so that they can be tested in a reusable test bed using a plug-in-and-play approach.***
- ***It is always a good idea to separate the component functional code from the added and built-in code that facilitates component testing and maintenance.***

Different Architecture Models for Testable Components



Maturity Levels for Testability

Evaluating the maturity levels of a test process concerning testability:

Level #1- Initial – At this level, component developers and testers use an ad hoc approach to enhance component testability in a component development process.

Level #2- Standardized – At this level, component testability requirements, design methods, implementation mechanisms, and verification criteria are defined as standards.

Level #3- Systematic – At this level, a well-defined component development and test process and systematic solutions are used to increase component testability at all engineering phases.

Level #4-Masurable – At this level, component testability can be evaluated and measured using systematic solutions and tools in all component development phases.

Verification of Component Testability

Verification of component testability:

→ Check component testability of software components using well-defined verification means during a component development process.

Static verification approach –

Using various verification methods to check the generated component artifacts in all phases, including component requirements, interface specifications, design logic, implementation, and test cases and results.

→ This enhances component testability by discovering testability issues in all phases of a component development process

Statistic verification approach –

Using statistical methods to analyze and estimate component testability by examining how a given component will behave when it contains faults.

→ This suggests the testing intensity or testing difficulty in discovering a fault at a specific location.

→ This suggests the number of tests necessary to gain quality confident.

Verification of Component Testability

Static verification approach –

→ **Component specification phase:**

Checking component requirements are clearly specified so that they can be tested and measured for a given test criteria.

How to specify them? How to verify them for testability?

→ **Component design phase:**

Checking component design for testability -> focusing how the current component design to meet the given testability requirements, including component model, architecture, interfaces for testing, test facility design

How to verify design artifacts for component testability?

→ **Component implementation phase:**

Checking if component design for testability has been properly implemented

→ **Component testing phase:**

Checking component tests based on the given test criteria

Measuring component testability based on a component testability model

Verification of Component Testability

Statistical verification approach –

→ **Use a statistical approach to examine how a given program behave when it contains a fault.**

Example: Jeffrey Voas proposed a verification approach (sensitivity analysis) to check program testability.

Its major objective is to predict the probability of a software failure occurring if the particular software contains a fault for a given set of test set for black-box testing.

Jerrfrey Voas' method involves three estimations at each location:

→ **Execution probability**

→ **Infection probability**

→ **Propagation probability**

Measurement of Software Testability

What is software testability measurement?

→ **Software testability measurement refers to the activities and methods that study, analyze, and measure software testability during a product development cycle.**

How to measure software testability? Three types of measurement methods:

- **Program-based measurement methods for software testability**

  Example, J. –C. Lin's program-based method to measure program testability by considering the single faults in a program.

- **Model-based measurement methods for software testability**

  Example, C. Robach and Y. Le Traon use the data flow model to measure software testability. Similarly, J.-C. Lin and S.-W. Lin's approach.

- **Dependability assessment methods for software testability**

  Example, A. Bertolino and L. Strigni's black-box approach which measures software testability based on the dependency relationships between inputs and corresponding outputs.

Measurement of Software Testability

Program-based measurement methods for software testability

Example, J. –C. Lin's program-based method to measure program testability by considering the single faults in a program.

The basic idea of this approach is similar to software mutation testing.

To compute the testability of a software at a specific location based on a single failure assumption:

- **A single fault is instrumented into the program at a specific location.**
- **The newly instrumented program is compiled and executed with an assumed input distribution.**
- **Three basic techniques (execution, infection, and propagation estimation) are used to compute the probability of failure that would occur when that location has a fault.**

Measurement of Software Testability

Model-based measurement methods for software testability

Example, C. Robach and Y. Le Traon use the data flow model to measure software testability. Similarly, J.-C. Lin and S.-W. Lin's approach.

-> A white-box based approach for testability measurement

Three steps:

1. **Normalizing a program before the testability measurement using a systematic tool.**
   - **Structure normalization and block normalization**

2. **Identifying the testable elements of the target program based on its normalized data flow model.**
   - **Including number of non-comment lines, nodes, edges, p-uses, defs, uses, d-u paths, and dominating paths.**

3. **Measuring the program testability based on data flow testing criteria.**
   - **Including ALL-NODES, ALL-EDGES, ALL-P-USES, ALL-DEFS, ALL-USES, ALL-DU-PAIRS and ALL-DOMINATING PATH.**

Measurement of Software Testability

Dependability assessment methods for software testability

Example, A. Bertolino and L. Strigni's black-box approach which measures software testability based on the dependency relationships between inputs and corresponding outputs.

- **A black-box approach for testability measurement**
- **Testability is computed based on the probability of a test of the program based on a given input setting is rejected by the program due to its faculty.**

The basic approach consists of the following steps:

- **Perform an oracle in a manual (or systematic) mode to decide whether a given program behave correctly on a given test.**
- **The oracle decides the test outcome by analyzing the behavior of the program against its specification.**
- **Observes the input and the output of each test against the expected output, and looks for failures.**