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.**

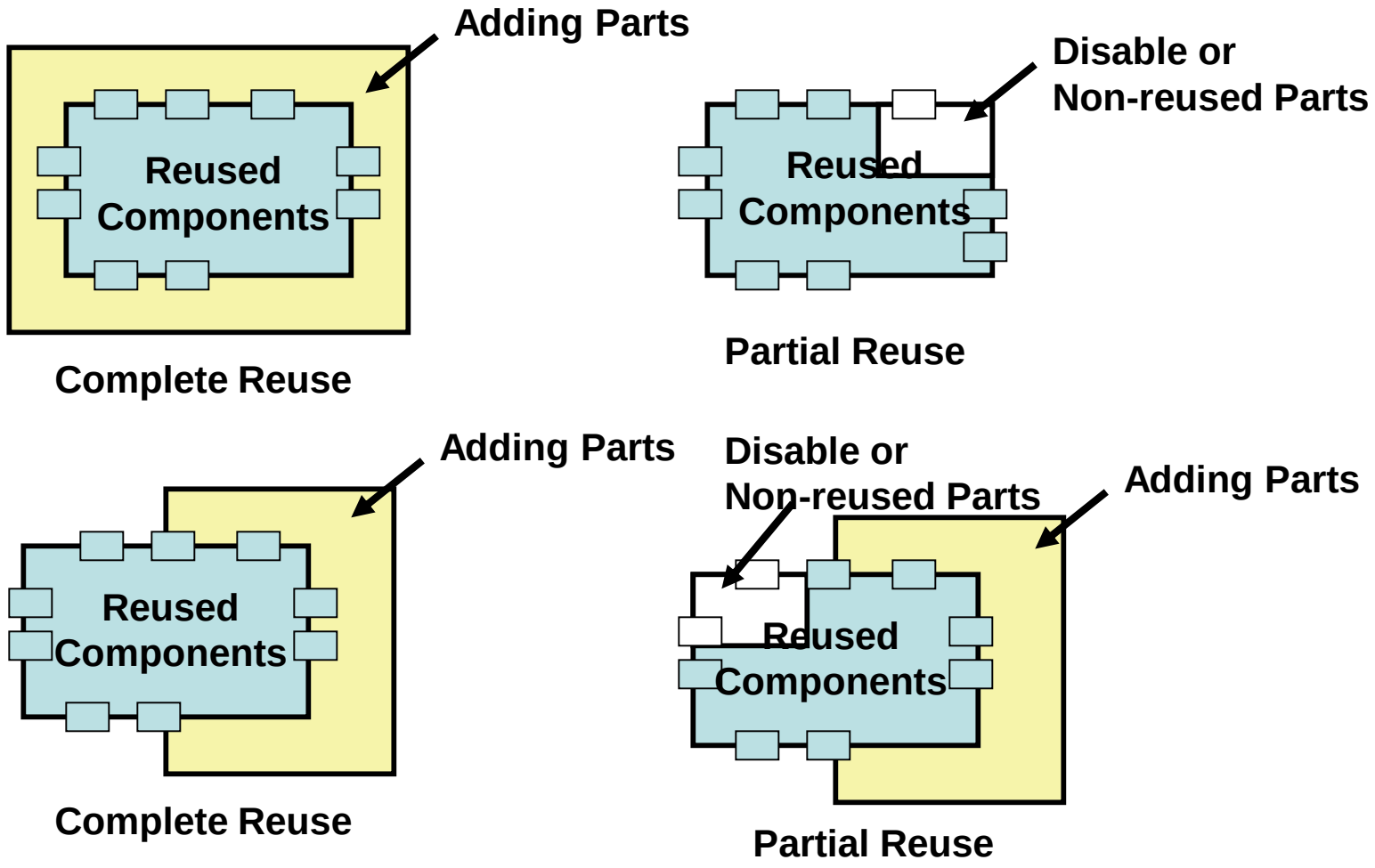
Software Component Validation

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)

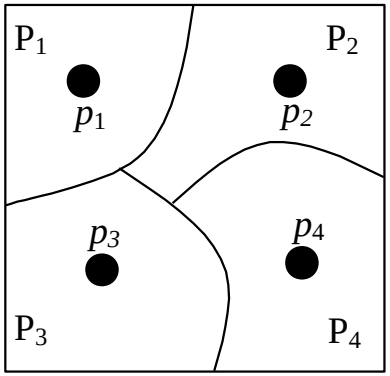
-File access environment
- File not accessible
- File can’t read
- File can’t open

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

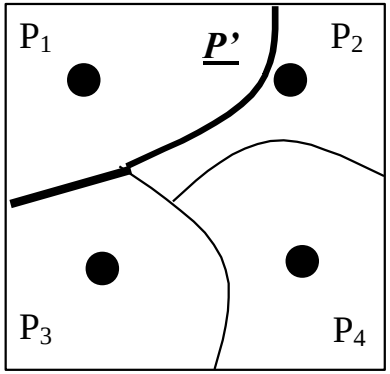
-Pattern occurrences on target line:
- one
- more than one
- None

Black-box Testing Methods for Components

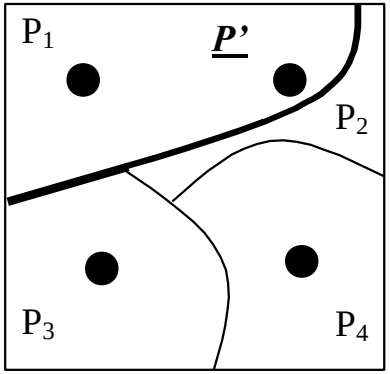
Partition Testing of Generalization Customization:



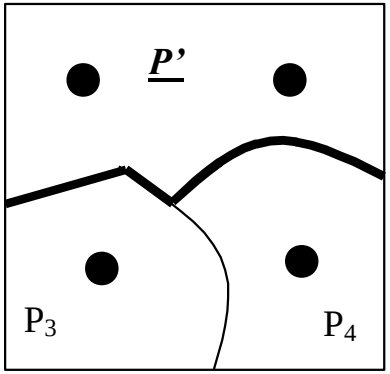
6.1 (a) Before generalization customization



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



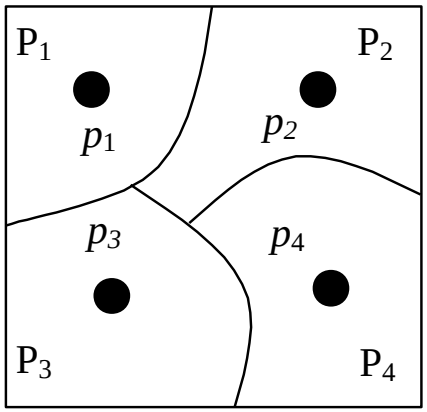
6.1 (c) Case (ii): After generalization customization



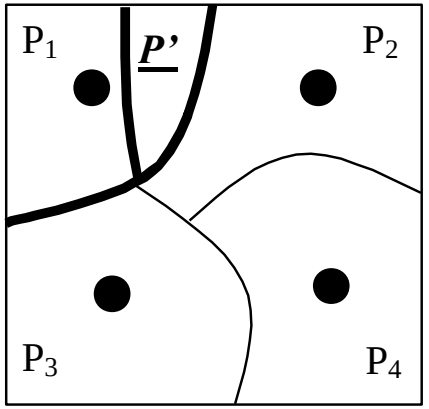
6.1 (d) Case (iii): After generalization customization

Black-box Testing Methods for Components

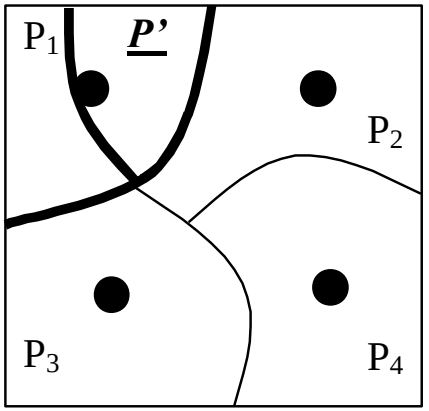
Partition Testing for Specialization Customization:



6.2 (a) Before specialization customization



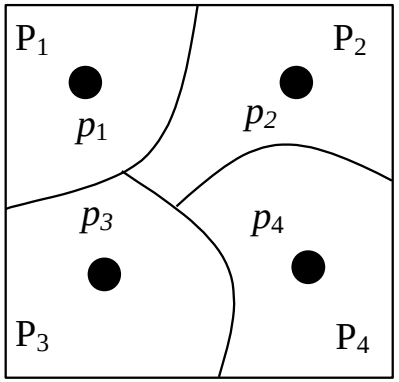
6.2 (b) Case (i): After specialization customization



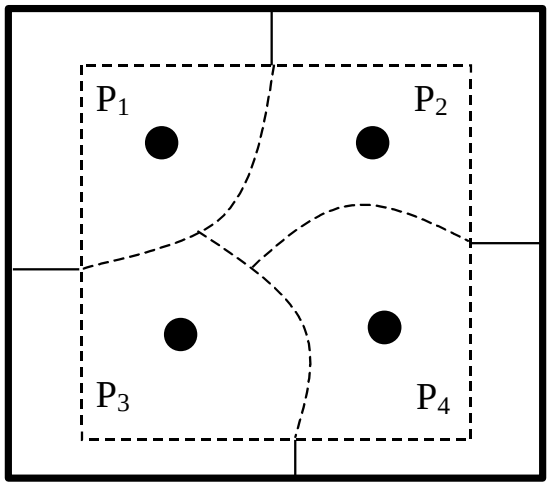
6.2 (c) Case (ii): After specialization customization

Black-box Testing Methods for Components

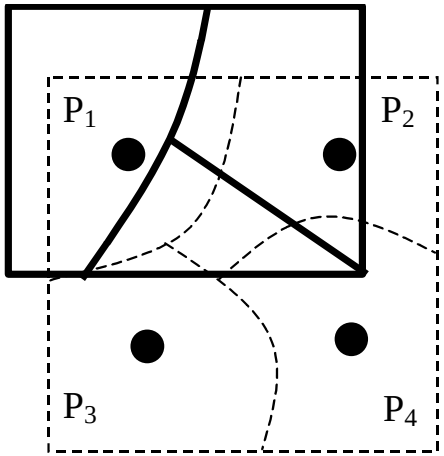
Partition Testing for Reconstruction Customization:



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:

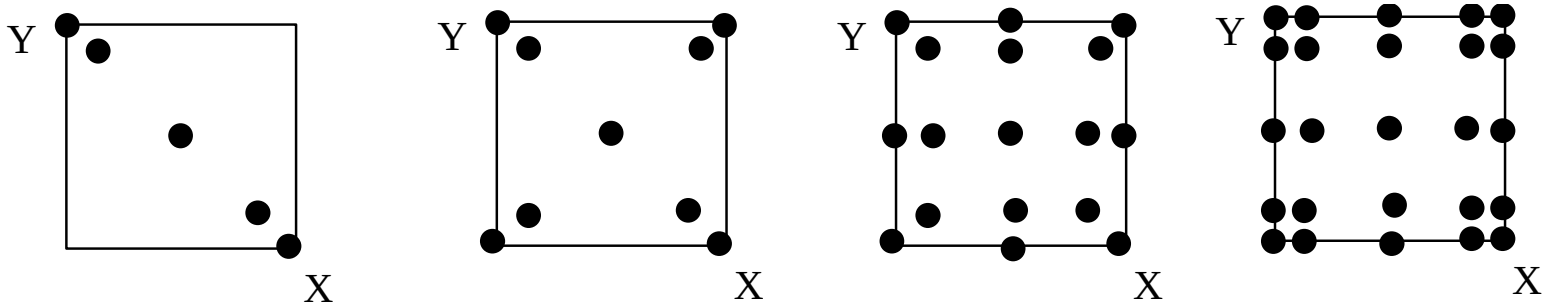
For each partition for an input domain, select boundary values for this partition to generate test cases.

Advantages of this approach:

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

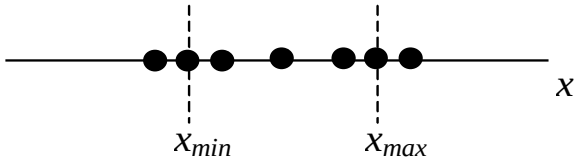
boundary values for one-dimensional space x in [X_{min}, X_{max}], y in [Y_{min}, Y_{max}]:

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}***

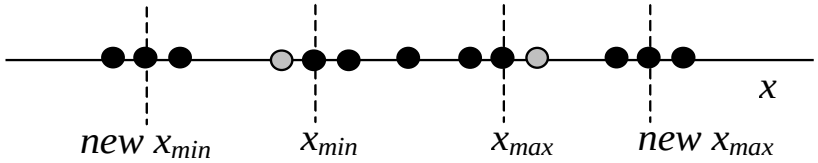


Black-box Testing Methods for Components

Boundary Value Testing:



6.5 (a) Before generalization customization

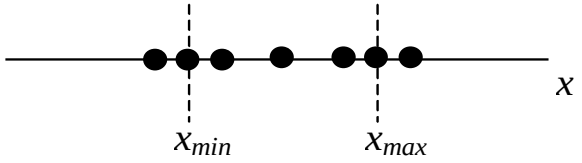


6.5 (b) After generalization customization

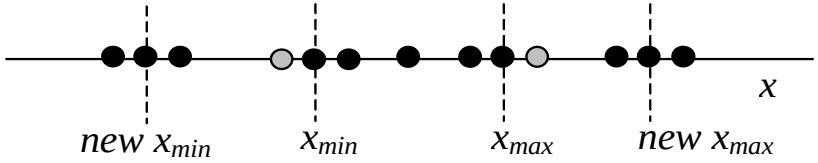
Black-box Testing Methods for Components

Boundary Value Testing – generalization customization:

→ Merge boundaries, add new boundaries and new boundary values



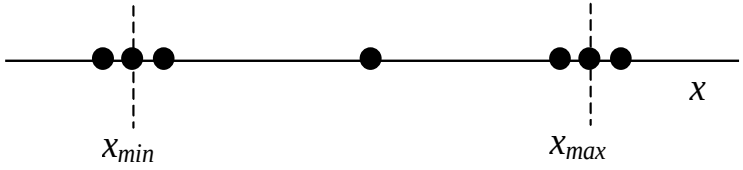
6.5 (a) Before
generalization customization



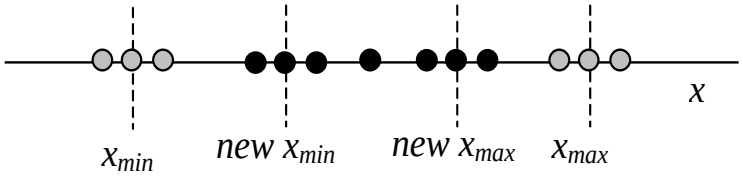
6.5 (b) After
generalization customization

Boundary Value Testing – generalization customization:

→ Add, change boundaries and new boundary values



6.6 (a) Before
specialization customization

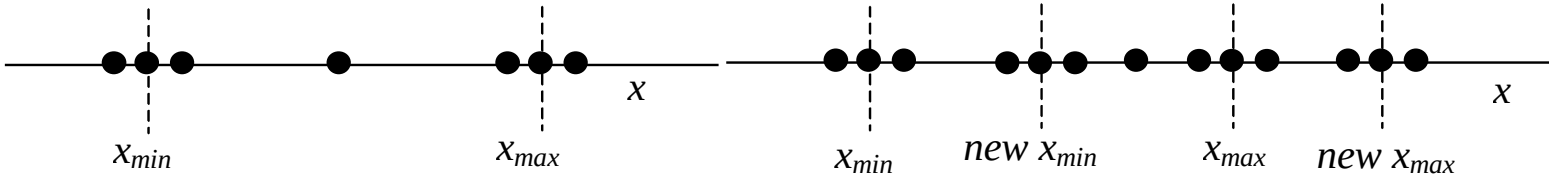


6.6 (b) After
specialization customization

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)

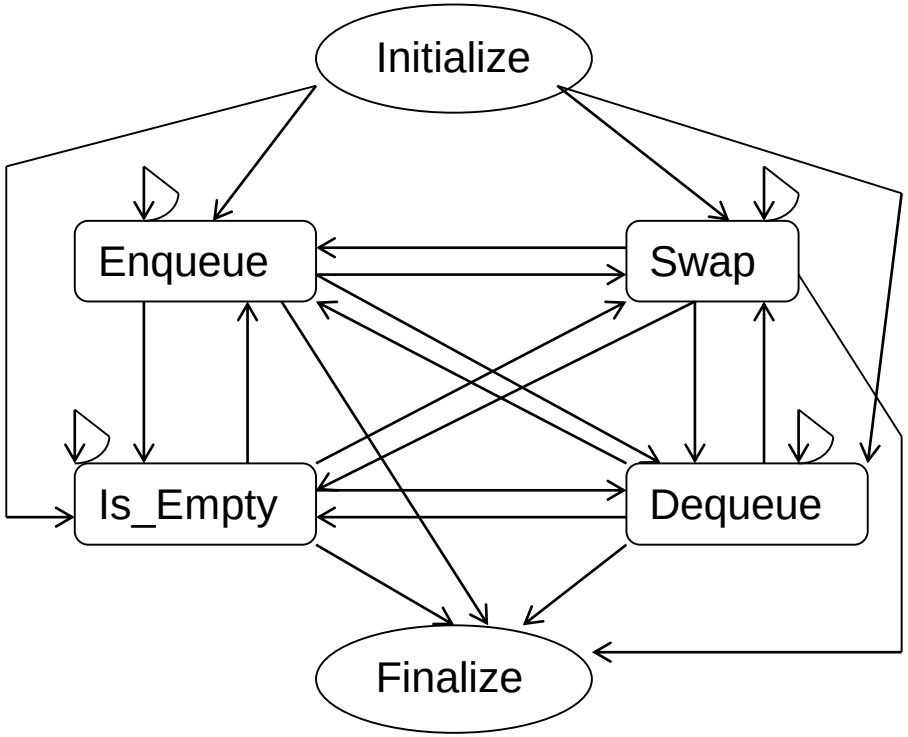


Figure 8.4. The Queue Flowgraph [6]

Black-box Testing Methods for Components

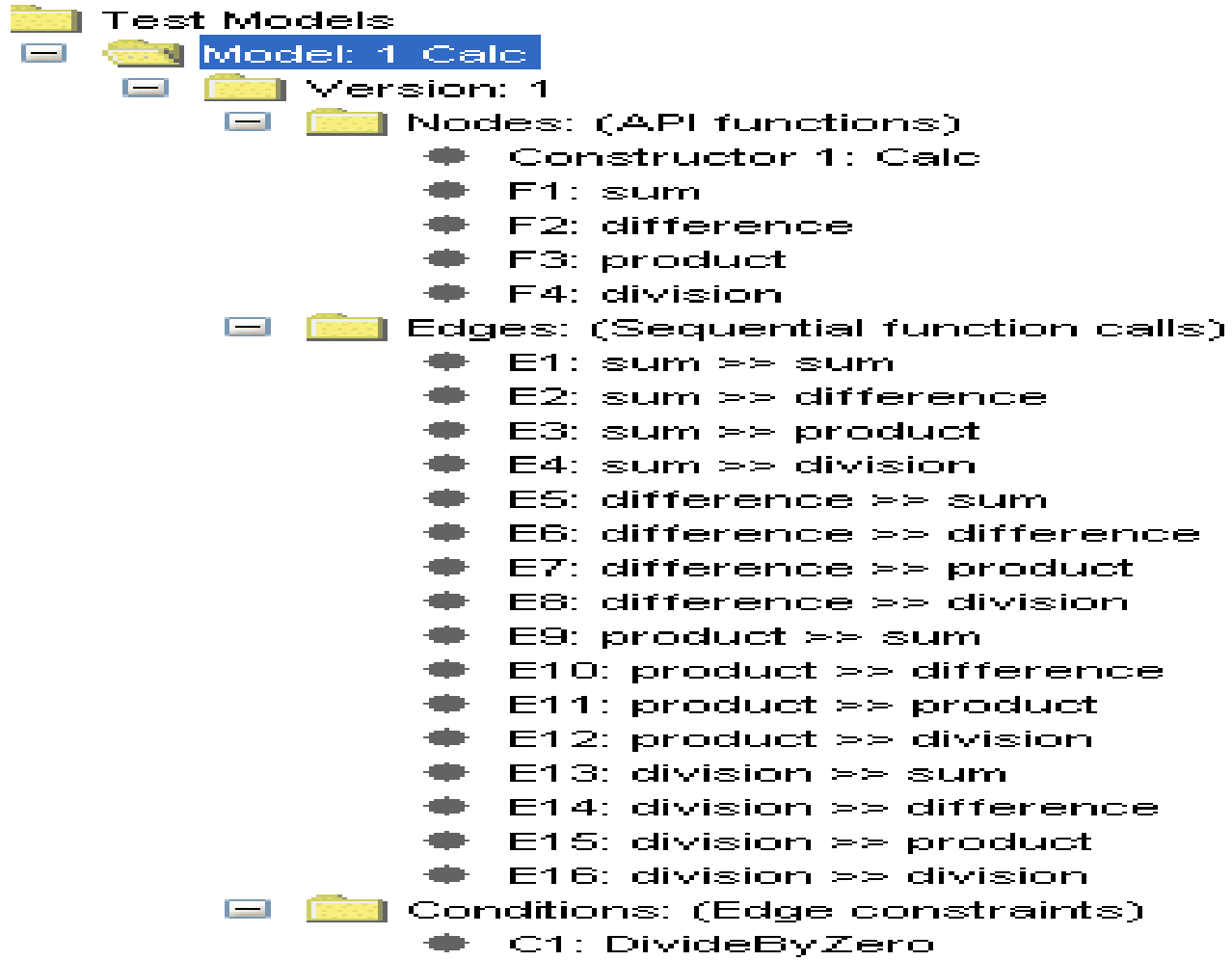
A RESOLVE queue specification:

```
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

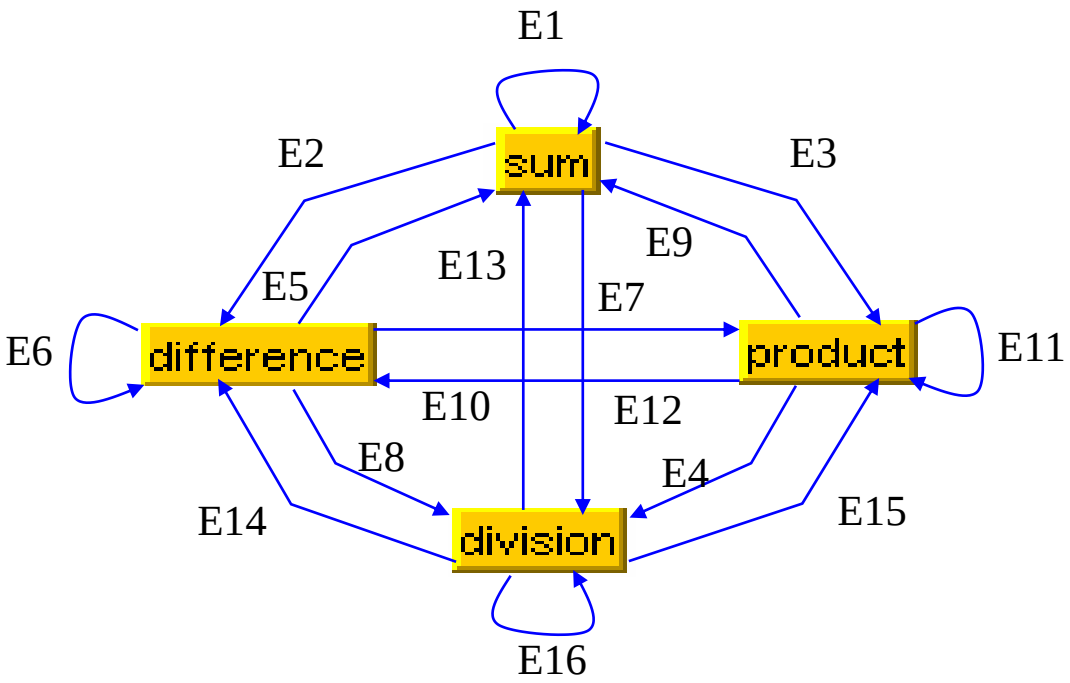


Figure 2. CFAG Test Model for a Calculator Component (Graphic Format)

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.***