วิศวกรรมซอฟต์แวร์

Software Engineering

สมเกียรติ วังศิริพิทักษ์

somkiat.wa@kmitl.ac.th

ห้อง 518 หรือ ห้อง 506 (MIV Lab)

PART II

Quality Management

Testing Conventional Applications

Deriving Test Cases

- The **basis path testing** method can be **applied to** a procedural design or to source code.
- In this section, we present basis path testing as **a series of steps**.
- The **procedure** average, depicted in PDL in the right figure, will be used as an example to illustrate each step in the test-case design method.
- The procedure average contains **compound conditions** and **loops**.
- The following **steps** can be applied to **derive the basis set**:

PDL with nodes identified

```
PROCEDURE average;
    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
         minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;                          2
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100         3
4     increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum      6
5       THEN increment total.valid by 1;
7               sum = s sum + value[i]
        ELSE skip
8     ENDIF
      increment i by 1;
9   ENDDO
    IF total.valid > 0                                      10
11    THEN average = sum / total.valid;
12      ELSE average = -999;
13  ENDIF
END average
```


Noted that V(G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1.

Basis Path Testing

Deriving Test Cases

1. Using the design or code as a foundation, **draw** a corresponding **flow graph**.

- A flow graph is created by numbering those PDL **statements** that will be **mapped into** corresponding flow graph **nodes**.

2. **Determine** the **cyclomatic complexity** of the resultant flow graph.

- The cyclomatic complexity *V(G)* is determined by applying the algorithms described earlier.

V(G) =

V(G) =

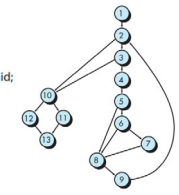
V(G) =

PROCEDURE average;

    * This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
         minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;   2
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100   3
4   increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5   THEN increment total.valid by 1;
7   sum = s sum + value[i]
        ELSE skip
8   ENDIF
      increment i by 1;
9   ENDDO
    IF total.valid > 0   10
11   THEN average = sum / total.valid;
12   ELSE average = -999;
13 ENDIF
END average



Deriving Test Cases

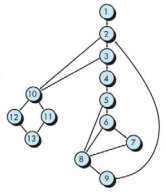
3. **Determine** a **basis set** of **linearly independent paths**.

- The value of *V(G)* provides the number of linearly independent paths through the program control structure.
- In the case of procedure average:  paths.

- It is often worthwhile to identify **predicate nodes** as an aid in the derivation of test cases.
  - In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

```
PROCEDURE average;

* This procedure computes the average of 100 or fewer
  numbers that lie between bounding values; it also computes the
  sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
   increment total.input by 1;
   IF value[i] > = minimum AND value[i] < = maximum
      THEN increment total.valid by 1;
           sum = s sum + value[i]
      ELSE skip
   ENDIF
   increment i by 1;
ENDDO
IF total.valid > 0
   THEN average = sum / total.valid;
   ELSE average = -999;
ENDIF
END average
```

PDL with nodes identified



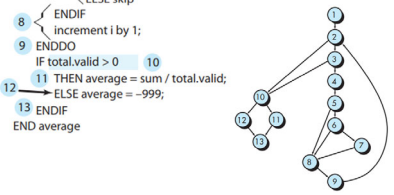
Deriving Test Cases

4. **Prepare test cases** that will force **execution** of **each path** in the basis set.

- Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
- Each test case is executed and compared to expected results.
- Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

```
PROCEDURE average;

* This procedure computes the average of 100 or fewer
  numbers that lie between bounding values; it also computes the
  sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
   increment total.input by 1;
   IF value[i] > = minimum AND value[i] < = maximum
      THEN increment total.valid by 1;
           sum = s sum + value[i]
      ELSE skip
   ENDIF
   increment i by 1;
ENDDO
IF total.valid > 0
   THEN average = sum / total.valid;
   ELSE average = -999;
ENDIF
END average
```

PDL with nodes identified

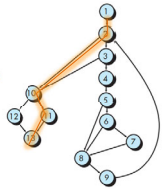


Deriving Test Cases

- It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion.
- That is, the combination of data required to traverse the path **cannot** be achieved **in the normal flow** of the program.
- In such cases, these paths are **tested as part of another path test**.

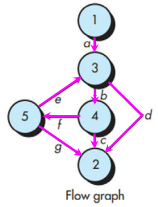
```
PROCEDURE average;

* This procedure computes the average of 100 or fewer
  numbers that lie between bounding values; it also computes the
  sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
   increment total.input by 1;
   IF value[i] > = minimum AND value[i] < = maximum
      THEN increment total.valid by 1;
           sum = s sum + value[i]
      ELSE skip
   ENDIF
   increment i by 1;
ENDDO
IF total.valid > 0
   THEN average = sum / total.valid;
   ELSE average = -999;
ENDIF
END average
```

PDL with nodes identified



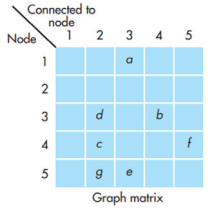
Graph Matrices

- A data structure, called a **graph matrix**, can be quite useful for developing a software tool that **assists** in **basis path testing**.
- A graph matrix is a **square matrix** whose **size** (i.e., number of rows and columns) is equal to the **number of nodes** on the flow graph.
- **Each row** and **column** corresponds to an identified **node**, and **matrix entries** correspond to connections (an **edge**) between nodes.



Flow graph

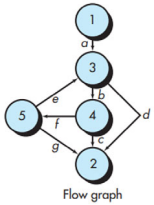
- A simple **example** of a flow graph and its corresponding graph matrix is shown in the figure.
  - Referring to the figure, each **node** on the flow graph is identified by numbers, while each **edge** is identified by letters
  - A letter entry is made in the matrix to correspond to a connection between two nodes.
  - For example, node 3 is connected to node 4 by edge b.



Graph matrix

Graph Matrices

- To this point, the graph matrix is nothing more than a **tabular representation** of **a flow graph**.
- However, by **add**ing a **link weight** to each matrix entry, the graph matrix can become a powerful **tool** for evaluating program control structure during testing.
- The link weight provides additional information about control flow.
- In its simplest form, the link weight is **1** (a connection exists) or **0** (a connection does not exist).



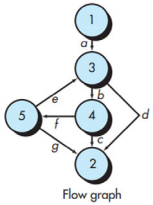
Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Graph Matrices

- But link weights can be assigned other, **more interesting properties**:
  - The probability that a link (edge) will be executed.
  - The processing time expended during traversal of a link
  - The memory required during traversal of a link
  - The resources required during traversal of a link.
- Using these techniques, the **analysis** required to design test cases can be partially or fully automated.



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

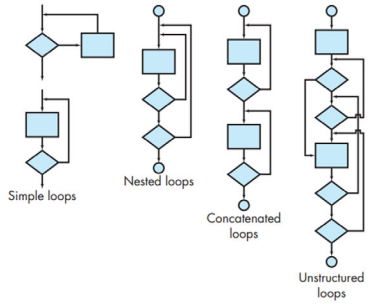
Control Structure Testing

- The basis path testing technique is one of a number of techniques for **control structure testing**. (it is not sufficient in itself.)
- **Other** variations on **control structure testing** will help improve the quality of white-box testing.

- ***Condition testing*** is a test-case design method that exercises the logical conditions contained in a program module.
- ***Data flow testing*** selects test paths of a program according to the locations of definitions and uses of variables in the program.

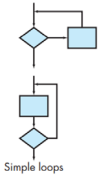
Control Structure Testing

- ***Loop testing*** is a white-box testing technique that focuses exclusively on the validity of loop constructs.
- Four different classes of loops can be defined:
  simple loops,
  concatenated loops,
  nested loops, and
  unstructured loops.



Control Structure Testing

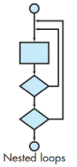
Loop Testing – Simple Loops



- The following **set of tests** can be applied to simple loops, where ***n*** is the **maximum number** of allowable **passes** through the loop.
  1. Skip the loop entirely.
  2. Only one pass through the loop.
  3. Two passes through the loop.
  4. *m* passes through the loop where $m < n$.
  5. *n*-1, *n*, *n*+1 passes through the loop.

Control Structure Testing

Loop Testing – Nested Loops



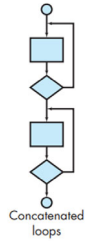
- The number of possible tests would grow geometrically as the level of nesting increases.
  - This would result in an impractical number of tests.
- An approach that will help to reduce the number of tests:
  1. Start at the innermost loop. Set all *other loops* to *minimum values*.
  2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values.

     Add other tests for *out-of-range* or *excluded values*.
  3. Work outward, conducting tests for the next loop, but keeping *all other outer loops* at *minimum values* and *other nested loops* to "*typical*" *values*.
  4. Continue until all loops have been tested.

min, min+1, typical, max-1, max

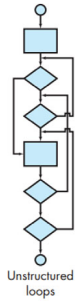
Control Structure Testing

Loop Testing – Concatenated Loops



- Concatenated loops can be tested using the **approach** defined for **simple loops**, if each of the loops is independent of the other.
- However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.
  → the **approach** applied to **nested loops** is recommended

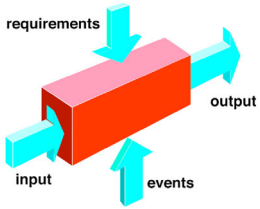
Loop Testing – Unstructured Loops



- Whenever possible, this class of loops should be **redesigned** to reflect the **use** of the **structured programming constructs**.

Black-Box Testing

Black-Box Testing



- Black-box testing, also called **behavioral testing** or **functional testing**, focuses on the *functional requirements* **of the software**.
- It is a ***complementary approach*** that is likely to *uncover* a *different class of errors than white-box methods*.
- Black-box testing attempts to find errors in the following **categories**:
  (1) incorrect or missing functions,
  (2) interface errors,
  (3) errors in data structures or external database access,
  (4) behavior or performance errors, and
  (5) initialization and termination errors.

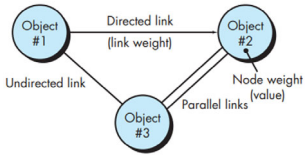
Black-Box Testing

- Unlike white-box testing, which is performed early in the testing process, **black-box testing** tends to be applied during **later stages** of testing.
- Because black-box testing purposely disregards control structure, attention is focused on the information domain.
- Tests are designed to answer the following questions:
  - How is functional validity tested?
  - How are system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?
  - What data rates and data volume can the system tolerate?
  - What effect will specific combinations of data have on system operation?

Graph-Based Testing Methods

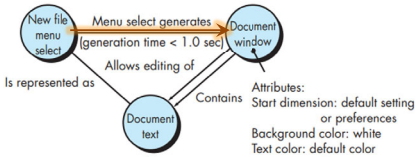
- The **first step** in black-box testing is to **understand the objects** that are modeled in software and the **relationships** that connect these objects.
- Once this has been accomplished, the **next step** is to define a **series of tests** that verify "all objects have the expected relationship to one another".
- Stated *in another way*,
  - software testing begins by **creating a graph** of important objects and their relationships
  - and then devising a **series of tests** that will cover the graph so that each object and relationship is exercised and errors are uncovered.

Graph-Based Testing Methods



- To accomplish these steps, you begin by creating a **graph**—a collection of ...
  - **nodes** that represent objects,
  - **links** that represent the relationships between objects,
  - **node weights** that describe the properties of a node (e.g., a specific data value or state behavior), and
  - **link weights** that describe some characteristic of a link.
- The symbolic representation of a graph is shown in the figure.
- **Nodes** are represented as **circles** connected by **links** (many different forms).
- A **directed link** (an **arrow**) indicates a relationship with **only one direction**.
- A **bidirectional link**, also called a **symmetric link**, implies that the relationship applies in **both directions**.
- **Parallel links** are used when **a number of different relationships** are established between graph nodes.

Graph-Based Testing Methods



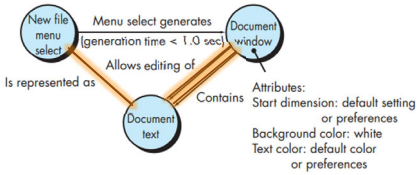
- As a simple **example**, consider a portion of a graph for a word-processing application where

  Object #1 = **newFile** (menu selection)

  Object #2 = **documentWindow**

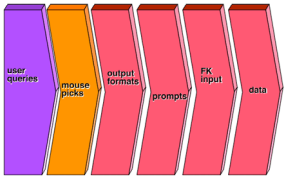
  Object #3 = **documentText**
- Referring to the figure, ...
- a **menu select** on **newFile** generates a document window.
- The **node weight** of **documentWindow** provides a list of the window that are to be expected when the window is generated.
- The **link weight** indicates that the window *must be generated in less than 1.0 second*.

Graph-Based Testing Methods



- An **undirected link** establishes a symmetric relationship between the **newFile** menu selection and **documentText**.
- **Parallel links** indicate relationships between **documentWindow** and **documentText**.
- **In reality**, a far more detailed graph would have to be generated as a precursor **to test-case design**.
  - You can then derive **test cases** by ***traversing the graph*** and covering each of the relationships shown.
  - These test cases are designed in an attempt to find errors in any of the relationships.

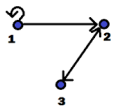
Equivalence Partitioning



- **Equivalence partitioning** is a ***black-box testing method*** that ***divides*** the ***input domain*** of a program ***into classes*** of data from which test cases can be derived.

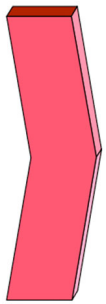
- An **ideal test case** **single-handedly** ***uncovers a class of errors*** (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.

- Test-case design for equivalence partitioning is based on an **evaluation** of ***equivalence classes*** **for an input condition**.
- Using concepts introduced in the preceding section, if a **set of objects** can be *linked by relationships* that are *symmetric*, *transitive*, and *reflexive*, an **equivalence class** is present.



Equivalence Partitioning

- An **equivalence class** represents a set of **valid** or **invalid states** for input conditions.
  - Typically, an input condition is either a ***specific numeric value***, a ***range*** of values, a ***set*** of related values, or a ***Boolean*** condition.



Valid data
- **user supplied commands**
- **responses to system prompts**
- **file names**
- **computational data**
  - **physical parameters**
  - **bounding values**
  - **initiation values**
- **output data formatting**
- **responses to error messages**
- **graphical data (e.g., mouse picks)**

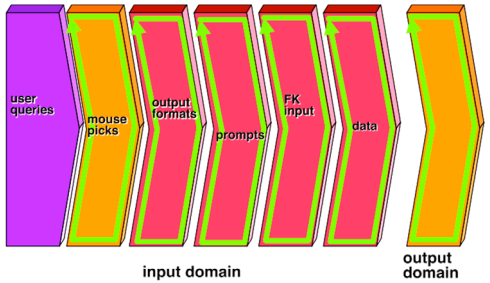
Invalid data
- **data outside bounds of the program**
- **physically impossible data**
- **proper value supplied in wrong place**

Besides the input condition, other conditions should also be paritiouned.

Equivalence Partitioning

- Equivalence classes may be defined according to the following **guidelines**:
  1. If an **input condition** specifies a **range**, one valid and two invalid equivalence classes are defined.
  2. If an input condition requires a **specific value**, one valid and two invalid equivalence classes are defined.
  3. If an input condition specifies a **member of a set**, one valid and one invalid equivalence class are defined.
  4. If an input condition is **Boolean**, one valid and one invalid class are defined.
- By applying the guidelines for the derivation of equivalence classes, **test cases** ***for each input domain data item*** can be **developed** and **executed**.
  - Test cases are **selected so that** the **largest number of attributes** of an equivalence class are **exercised at once**.

Boundary Value Analysis



- A greater number of **errors** occurs at the **boundaries** of the input domain rather than in the "center."
- **Boundary value analysis** (BVA) leads to a selection of test cases that exercise bounding values.
- Boundary value analysis is a test-case design technique that **complements** equivalence partitioning.
  - Rather than selecting any element of an equivalence class, BVA leads to the **selection** of test cases at the "**edges**" of the class.
  - Rather than focusing solely on input conditions, BVA **derives** test cases **from** the **output domain** ***as well***.

Boundary Value Analysis

- **Guidelines for BVA** are similar in many respects to those provided for equivalence partitioning:
  1. If an **input condition** specifies a **range** bounded by values ***a*** **and** ***b***, test cases should be designed with values *a* and *b* and just above and just below a and b.
  2. If an input condition specifies **a number of values**, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
  3. Apply guidelines 1 and 2 to **output conditions**.
     (E.g., assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.)
  4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Comparison Testing

- Used **only** in situations in which the **reliability of software is absolutely critical** (e.g., ***human rated systems***)
  - Separate software engineering teams **develop** ***independent versions*** **of an application** using the ***same specification***
  - Each version can be ***tested with the same test data*** to ***ensure*** that all provide ***identical output***
  - Then ***all versions are executed in parallel*** with ***real-time comparison of results*** to ***ensure consistency***

ANOTHER MEANING
Comparison testing is a testing technique that evaluates and **compares** a software product's strengths and weaknesses **to other similar products in the market**.

Orthogonal Array Testing

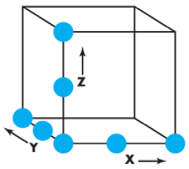
- There are many applications in which the **number of input parameters** is **small** and the ***values*** that each of the parameters may take are clearly ***bounded***.
  - When these numbers are very small (e.g., *three* input *parameters* taking on *three* discrete *values* each), it is possible to consider every input permutation and exhaustively test the input domain.
  - However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

Note that the test mentioned here is for the parameter values only.
There are still **other tests** such as paths must be performed.

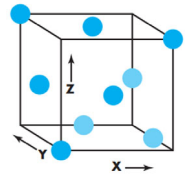
Orthogonal Array Testing

- Orthogonal array testing can be applied to **problems** in which the **input domain** is **relatively small** but too large to accommodate exhaustive testing.
  - The orthogonal array testing method is particularly useful in finding **region faults**—an error category associated with faulty logic *within* a software *component*.
- To illustrate the **difference** between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has ***three input items***, X, Y, and Z.
- **Each** of these input items **has three discrete values** associated with it.
- There are  possible test cases.
- Phadke suggests a **geometric view** of the possible test cases associated with X, Y, and Z illustrated in the next figure.

Orthogonal Array Testing



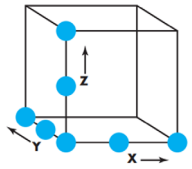
One input item at a time



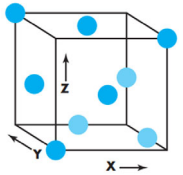
L9 orthogonal array

- Referring to the figure, one input item at a time may be varied in sequence along each input axis.
- This results in relatively limited coverage of the input domain (represented by the upper cube in the figure).
- When **orthogonal array testing** occurs, an L9 orthogonal array of test cases is created.
- The L9 orthogonal array has a "balancing property".
- That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain". (the bottom cube in the figure.)
- Test coverage across the input domain is more complete.

Orthogonal Array Testing



One input item at a time



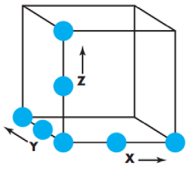
L9 orthogonal array

- To illustrate the use of the L9 orthogonal array, consider the **send function** for a **fax application**.
- **Four parameters**, P1, P2, P3, and P4, are passed to the send function.
- Each takes on **three** discrete **values**.
- For example, P1 takes on values:

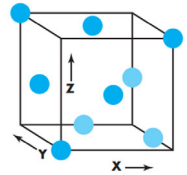
  P1 = 1, send it now
  P1 = 2, send it one hour later
  P1 = 3, send it after midnight

  P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.

Orthogonal Array Testing



One input item at a time



L9 orthogonal array

- If a "one input item at a time" testing strategy were chosen, the **following sequence of tests** (P1, P2, P3, P4) would be specified:

  (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1),
  (1, 2, 1, 1), (1, 3, 1, 1),
  (1, 1, 2, 1), (1, 1, 3, 1),
  (1, 1, 1, 2), and (1, 1, 1, 3).

- But these would **uncover** only single mode faults, that is, faults that are triggered by a single parameter.

Orthogonal Array Testing

- Given the **relatively small number** of **input parameters** and discrete values, exhaustive testing is possible.
- The number of tests required is  , **large** but **manageable**.
- All **faults** associated with data item permutation **would be found**, **but** the **effort** required is **relatively high**.
- The **orthogonal array testing** approach enables you to provide **good test coverage** with far **fewer test cases** than the exhaustive strategy.
- An L9 orthogonal array for the fax send function is illustrated in the Table.

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Orthogonal Array Testing

- Phadke **assesses** the **result of tests** using the **L9 orthogonal array** in the following manner:
- **Detect and isolate all single mode faults.** A single mode fault is a consistent problem with any level of any single parameter.
- For example, if all test cases of factor P1 = 1 cause an error condition, it is a **single mode failure**.
- In this example **tests 1, 2 and 3** will show **errors**.
- By analyzing the information about which tests show errors, one can identify **which parameter** values **cause the fault**.
- In this example, by noting that tests 1, 2, and 3 cause an error, one **can isolate** [logical processing associated with "send it now" (P1 = 1)] as the **source of the error**.
- Such an **isolation** of fault is **important to fix** the fault.

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Orthogonal Array Testing

- **Detect all double mode faults.** If there exists a consistent problem when specific levels of **two parameters occur together**, it is called a double mode fault.
- Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.
- Multimode faults. Many multi-mode faults are also detected by these tests.
- You can find a **detailed** discussion of orthogonal array testing in "Phadke, M., Quality Engineering Using Robust Design, Prentice Hall, 1989".

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Model-Based Testing

- Model-based testing (MBT) is a **black-box testing technique** that uses **information** contained **in the requirements model** as the basis **for the generation of test cases**.
  - In many cases, the model-based testing technique uses **UML state diagrams**, an element of the behavioral model, as the basis for the design of test cases.
- The MBT technique requires five steps:

1. **Analyze** an existing **behavioral model** for the software or create one.
   - Recall that a ***behavioral model*** indicates **how software** will **respond** to external events or stimuli.

Black-Box Testing

Model-Based Testing

2. **Traverse** the behavioral **model** and **specify** the inputs that will force the software **to make the transition** from state to state.
   - The inputs will trigger events that will cause the transition to occur.
3. **Review** the behavioral model and note the **expected outputs** as the software makes the transition from state to state.
4. **Execute** the **test cases**.
   - Tests can be executed ***manually*** or a test script can be created and executed using a *testing tool*.
5. **Compare actual** and **expected results** and take **corrective action** as required.

MBT helps to uncover **errors in software behavior**.
It is extremely **useful** when testing **event-driven applications**.

Testing Documentation and Help Facilities

- Nothing is more **frustrating** than following a user guide or an online help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation.
- It is for this reason that **documentation testing** should be a **meaningful** part of every software test plan.

- Documentation testing can be approached in **two phases**.
  - The **first phase**, technical review,
    examines the document *for* editorial *clarity*.
  - The **second phase**, live test,
    uses the documentation in *conjunction with* the actual *program*.

Testing Documentation and Help Facilities

- Surprisingly, a **live test** for documentation can be approached using **techniques** that are **analogous to** many of the **black-box testing methods** discussed earlier.
  - Graph-based testing can be used to ...
    describe the ***use*** of the program.
  - Equivalence partitioning and boundary value analysis can be used to ...
    define various ***classes*** of input and associated interactions.
  - MBT can be used to ...
    ensure that documented ***behavior*** and actual behavior coincide.
  - Program usage is then *tracked* through the documentation.

Testing for Real-Time Systems

- The **time-dependent**, **asynchronous** nature of many real-time applications adds a new and potentially difficult element to the testing mix—time.
- Not only does the test-case designer have to consider **conventional test cases ...**
  but **also event handling** (i.e., interrupt processing),
  the **timing of the data**,
  and the **parallelism of the tasks** (processes) that handle the data.
- In many situations,
  test data provided when a real-time system is in one state
  will result in proper processing,
  while the same data provided when the system is in a different state
  may lead to error.

Testing for Real-Time Systems

- For **example**, the real-time software that controls a new **photocopier** accepts ***operator interrupts*** (i.e., the machine operator ***hits control keys*** such as RESET or DARKEN) with no error **when** the machine is **making copies** (in the copying state).
  - These ***same operator interrupts***, if input when the machine is **in the jammed state**, cause a display of the diagnostic code indicating the location of the jam to be lost (an error).
- In addition, software tests must consider the **impact of hardware faults on software processing**.
  - Such faults can be extremely **difficult to simulate realistically**.

Testing for Real-Time Systems

- An overall **four-step strategy** for real-time software testing can be proposed:

❶ • **Task testing.**
Test each task **independently**.

- Task testing *uncovers errors* in ***logic*** and ***function*** but not timing or behavior.

Testing for Real-Time Systems

❷ • **Behavioral testing.**
Using system models created **with automated tools**, it is possible to ***simulate*** the ***behavior of a real-time system*** and examine its behavior as a consequence of external events.

- For **example**, events for the photocopier might be ...
  user interrupts (e.g., reset counter),
  mechanical interrupts (e.g., paper jammed),
  system interrupts (e.g., toner low), and
  failure modes (e.g., roller overheated).
- **Each** of these events is **tested individually**, and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events.

Testing for Real-Time Systems

❸ • **Intertask testing.**
Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors.

- Asynchronous tasks that are known to communicate with one another are tested with **different data rates** and **processing load** to determine if intertask **synchronization errors** will occur.
- In addition, tasks that communicate via a **message queue** or **data store** are tested to uncover **errors in** the **sizing** of these data storage areas.

Testing for Real-Time Systems

❹ • **System testing**. SW and HW are integrated, and a full range of system tests are conducted to uncover **errors** at the **software-hardware interface**.

- *Most real-time systems* ***process interrupts***.
- Using the state diagram, the tester develops a list of all **possible interrupts** and the processing that occurs as a consequence of the interrupts.
- **Tests** are then **designed** to **assess** the **following system characteristics**:

Patterns for Software Testing

- **Testing patterns** are described in much the same way as design patterns.
  - Dozens of testing patterns have been proposed in the literature.
- The following **three testing patterns** (presented in abstract form only) provide representative **examples**:
- *Pattern name*: **PairTesting**
- *Abstract*: A **process-oriented pattern**, pair testing describes a technique that is analogous to pair programming in which **two testers work together** to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

Patterns for Software Testing

- *Pattern name*: **SeparateTestInterface**
- *Abstract*: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The SeparateTestInterface pattern describes how to create "a test interface that can be used to describe **specific tests** on **classes** that are **visible only internally** to a component".
- *Pattern name*: **ScenarioTesting**
- *Abstract*: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The ScenarioTesting pattern describes a technique for **exercising** the software **from** the **user's point of view**. A failure at this level indicates that the software has failed to meet a user visible requirement.