دانشگاه علم و صنعت ایران

دانشکده مهندسی کامپیوتر

عنوان درس:

طراحی نرم‌افزارهای اتکاءپذیر

(Dependable Software Design)

فصل ۷: افزونگی نرم‌افزاری

مدرس: محمد عبداللهی ازگمی

(Mohammad Abdollahi Azgomi)

azgomi@iust.ac.ir

Software Redundancy

- **Reference:**
  - □ **E. Dubrova, *Fault-Tolerant Design: An Introduction*, Kluwer Academic Publisher (2007)**
    - **Chapter 7: Software Redundancy**
- **---**
- *Programs are really not much more than the programmer's best guess about what a system should do.*

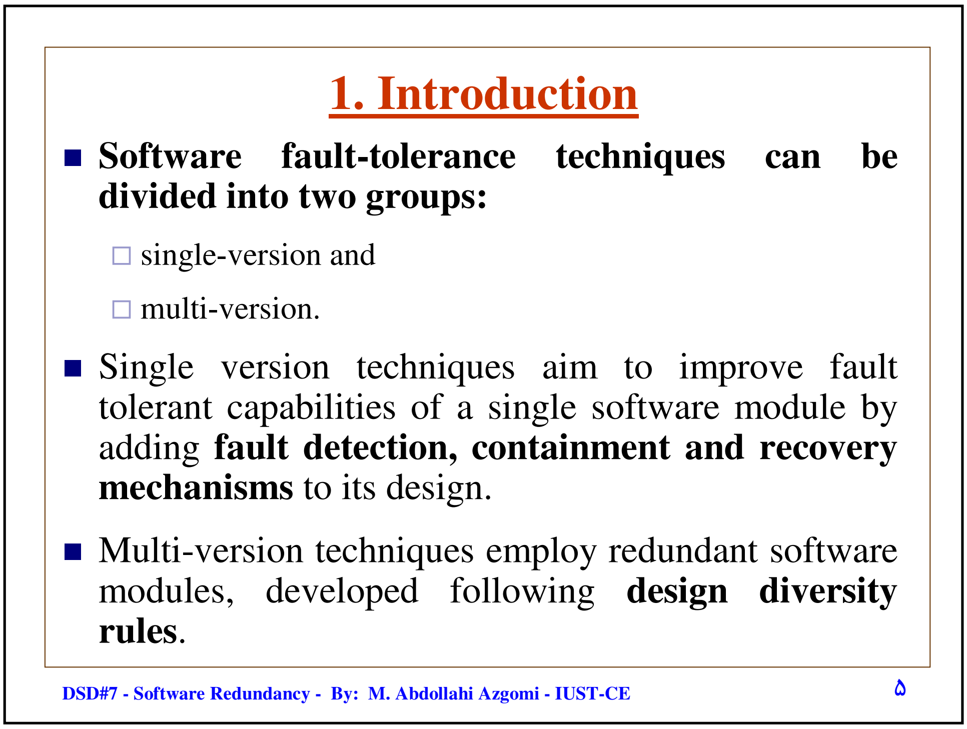
  —Russel Abbot

Contents

- 1. Introduction
- 2. Single-version techniques
  - □ Fault detection techniques
  - □ Fault containment techniques
  - □ Fault recovery techniques
- 3. Multi-version techniques
  - □ Recovery blocks
  - □ *N*-version programming
  - □ *N* self-checking programming
  - □ Design diversity
- 4. Software Testing
  - □ Statement Coverage
  - □ Branch Coverage
  - □ Branch Coverage

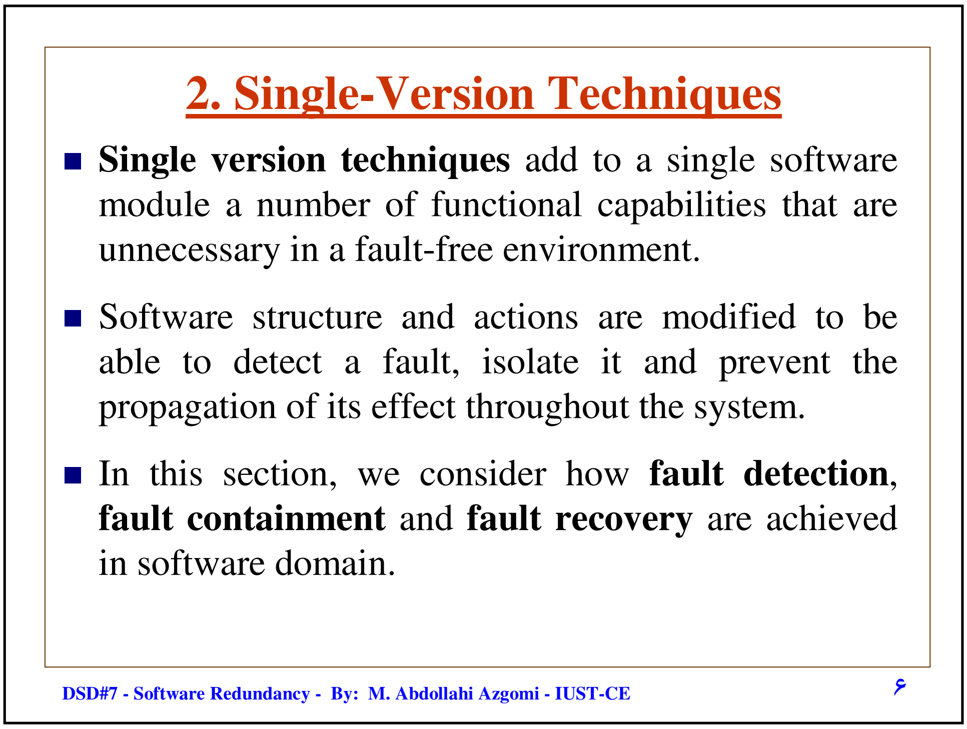
1. Introduction

- In this chapter, we discuss **techniques for software fault-tolerance**.
- **In general, fault-tolerance in software domain is not as well understood and mature as fault-tolerance in hardware domain.**



1. Introduction

- **Software fault-tolerance techniques can be divided into two groups:**
  - □ single-version and
  - □ multi-version.
- Single version techniques aim to improve fault tolerant capabilities of a single software module by adding **fault detection, containment and recovery mechanisms** to its design.
- Multi-version techniques employ redundant software modules, developed following **design diversity rules**.



2. Single-Version Techniques

- **Single version techniques** add to a single software module a number of functional capabilities that are unnecessary in a fault-free environment.
- Software structure and actions are modified to be able to detect a fault, isolate it and prevent the propagation of its effect throughout the system.
- In this section, we consider how **fault detection**, **fault containment** and **fault recovery** are achieved in software domain.

Fault Detection Techniques

- As in the hardware case, the goal of fault detection in software is **to determine that a fault has occurred within a system**.
- Single-version fault tolerance techniques usually use various types of ***acceptance tests*** to detect faults.
- The result of a program is subjected to a test.
  - □ **If the result passes the test, the program continues its execution. A failed test indicates a fault.**

Fault Detection Techniques

- A test is most effective if it can be calculated in a simple way and if it is based on **criteria** (مجموعه معیارها) that can be derived independently of the program application.
- The existing techniques include:
  - □ timing checks (بررسیهای تنظیم وقت),
  - □ coding checks (بررسیهای کدگذاری),
  - □ reversal checks (بررسیهای برگشت),
  - □ reasonableness checks (بررسیهای معقول بودن) and
  - □ structural checks (بررسیهای ساختاری).

Fault Detection Techniques

- ***Timing checks*** are applicable to systems whose specification include **timing constrains (قیود زمانی)**.
- Based on these constrains, checks can be developed to indicate a deviation from the required behavior.
  - □ ***Watchdog timer*** is an example of a timing check.
  - □ Watchdog timers are used to monitor the performance of a system and detect lost or locked out modules.

Fault Detection Techniques

- ***Coding checks*** are applicable to systems whose data can be encoded using information redundancy techniques.
  - □ Cyclic redundancy checks (CRC) can be used in cases when the information is merely transported from one module to another without changing it content.
  - □ **Arithmetic codes** can be used to detect errors in arithmetic operations.

Fault Detection Techniques

- In some systems, it is possible to reverse the output values and to compute the corresponding input values. For such system, ***reversal checks*** can be applied.
- A reversal check compares the actual inputs of the system with the computed ones. A disagreement indicates a fault.

Fault Detection Techniques

- ***Reasonableness checks*** use semantic properties of data to detect fault.
  - □ For example, a range of data can be examined for overflow or underflow to indicate a deviation from system's requirements.

Fault Detection Techniques

- ***Structural checks*** are based on known properties of data structures.
  - □ For example, a number of elements in a list can be counted, or links and pointers can be verified.
  - □ Structural checks can be made more efficient by adding redundant data to a data structure, e.g. attaching counts on the number of items in a list, or adding extra pointers.

Fault Containment Techniques

- **Fault containment** (تحدید خطا) in software can be achieved by modifying the structure of the system and by putting a set of restrictions defining which actions are permissible within the system.

Fault Containment Techniques

- In this section, we describe four techniques for fault containment:
  - □ **modularization,**
  - □ **partitioning,**
  - □ **system closure (محصور کردن سیستم) and**
  - □ **atomic actions.**

Fault Containment Techniques

- It is common to decompose a software system into *modules* with few or no common dependencies between them.
- **Modularization** attempts to prevent the propagation of faults by limiting the amount of communication between modules to carefully monitored messages and by eliminating shared resources.

Fault Containment Techniques

- Before performing modularization, ***visibility*** and ***connectivity*** parameters are examined to determine which module possesses highest potential to cause system failure.
  - ***Visibility*** of a module is characterized by the set of modules that may be invoked directly or indirectly by the module.
  - ***Connectivity*** of a module is described by the set of modules that may be invoked directly or used by the module.

Fault Containment Techniques

- The isolation between functionally independent modules can be done by ***partitioning*** the modular hierarchy of a software architecture in **horizontal** or **vertical** dimensions.
- **Horizontal partitioning** separates the major software functions into independent branches.
  - The execution of the functions and the communication between them is done using control modules.
- **Vertical partitioning** distributes the control and processing function in a top-down hierarchy.
  - Highlevel modules normally focus on control functions, while low-level modules perform processing.

Fault Containment Techniques

- Another technique used for fault containment in software is ***system closure***.
  - □ This technique is based on a principle that no action is permissible unless explicitly authorized.
  - □ In an environment with many restrictions and strict control (e.g. in prison) all the interactions between the elements of the system are visible.
  - □ Therefore, it is easier to locate and remove any fault.

Fault Containment Techniques

- An alternative technique for fault containment uses ***atomic actions*** to define interactions between system components.
- An atomic action among a group of components is an activity in which the components interact exclusively with each other.
- There is no interaction with the rest of the system for the duration of the activity.

Fault Containment Techniques

- Within an atomic action, the participating components neither import, nor export any type of information from non-participating components of the system.
- There are two possible outcomes of an atomic action:
  - □ **either it terminates normally, or**
  - □ **it is aborted upon a fault detection.**
  - □ If an atomic action terminates normally, its results are correct.
  - □ If a fault is detected, then this fault affects only the participating components.

Fault Recovery Techniques

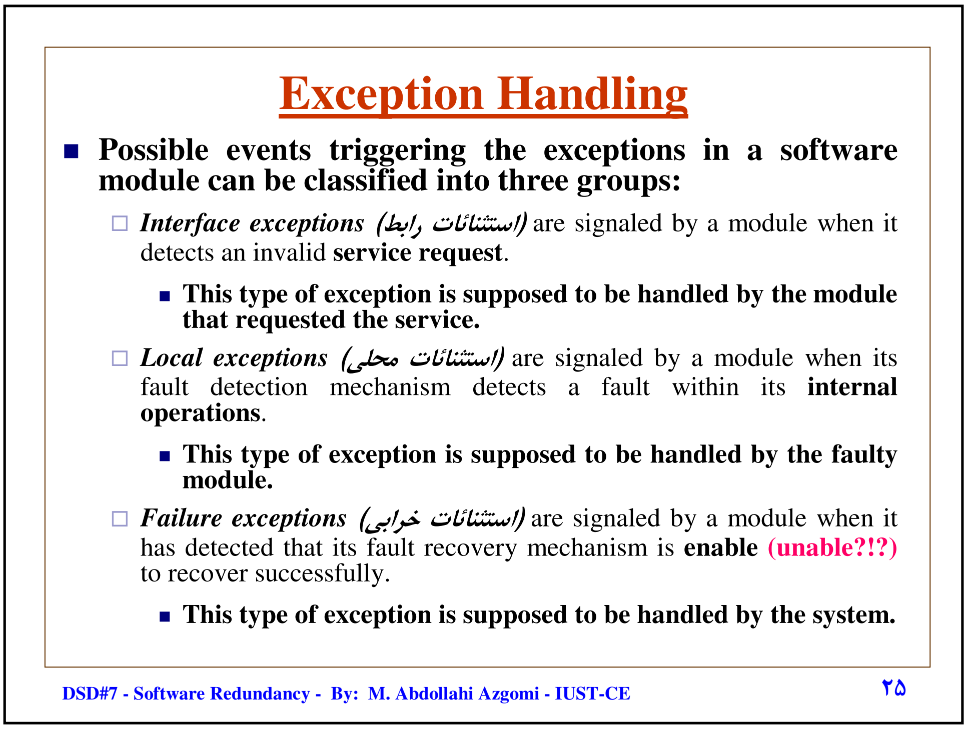
- Once a fault is detected and contained, a system attempts to **recover** from the faulty state and regain operational status.
- If fault detection and containment mechanisms are implemented properly, the effects of the faults are contained within a particular set of modules at the moment of fault detection.
- The knowledge of fault containment region is essential for the design of effective fault recovery mechanism.

Fault Recovery Techniques

- The following F.R. techniques will be discussed:
  - □ **Exception handling (مدیریت استثنائات),**
  - □ **Checkpoint and restart**
  - □ **Process pairs**
  - □ **Data diversity (تنوع طراحی)**

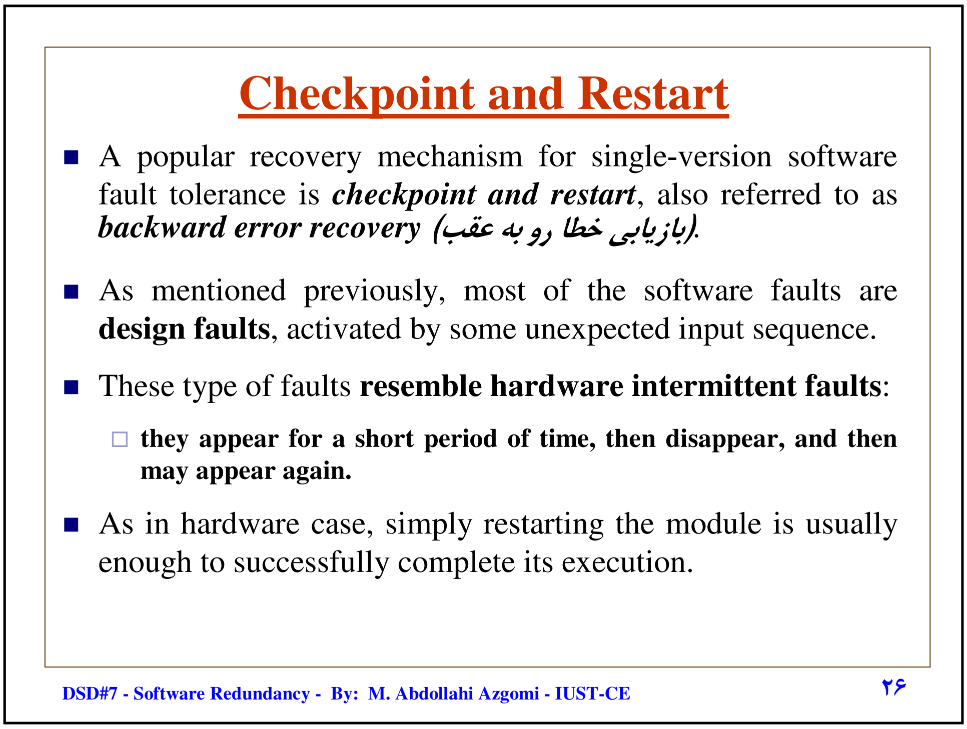
Exception Handling

- In many software systems, the request for initiation of fault recovery is issued by ***exception handling***.
  - □ **Exception handling is the interruption of normal operation to handle abnormal responses.**



Exception Handling

- **Possible events triggering the exceptions in a software module can be classified into three groups:**
  - □ ***Interface exceptions (استثنائات رابط)*** are signaled by a module when it detects an invalid **service request**.
    - **This type of exception is supposed to be handled by the module that requested the service.**
  - □ ***Local exceptions (استثنائات محلی)*** are signaled by a module when its fault detection mechanism detects a fault within its **internal operations**.
    - **This type of exception is supposed to be handled by the faulty module.**
  - □ ***Failure exceptions (استثنائات خرابی)*** are signaled by a module when it has detected that its fault recovery mechanism is **enable (unable?!?)** to recover successfully.
    - **This type of exception is supposed to be handled by the system.**



Checkpoint and Restart

- A popular recovery mechanism for single-version software fault tolerance is ***checkpoint and restart***, also referred to as ***backward error recovery*** *(بازیابی خطا رو به عقب).*

- As mentioned previously, most of the software faults are **design faults**, activated by some unexpected input sequence.

- These type of faults **resemble hardware intermittent faults**:
  - □ **they appear for a short period of time, then disappear, and then may appear again.**

- As in hardware case, simply restarting the module is usually enough to successfully complete its execution.

Checkpoint and Restart

- The general scheme of checkpoint and restart recovery mechanism is shown in Figure 7.1.

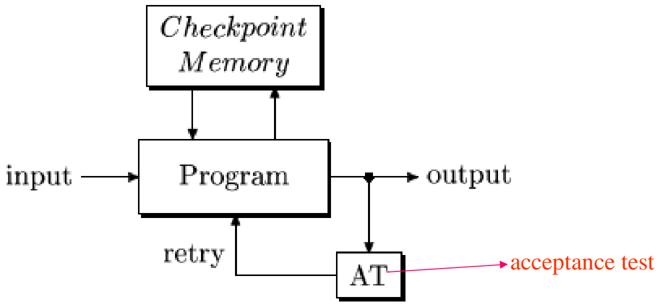


Figure 7.1. Checkpoint and restart recovery.

Checkpoint and Restart

- The module executing a program operates in combination with an **acceptance test block AT,** which checks the correctness of the result.
- If a fault is detected, a “retry” signal is send to the module to re-initialize its state to the checkpoint state stored in the memory.

Checkpoint and Restart

- **There are two types of checkpoints:**
  - □ **static and**
  - □ **dynamic.**

Checkpoint and Restart

- A **static checkpoint** takes a single snapshot of the system state at the beginning of the program execution and stores it in the memory.
  - □ **Fault detection checks are placed at the output of the module.**
  - □ **If a fault is detected, the system returns to this state and starts the execution from the beginning.**

Checkpoint and Restart

- **Dynamic checkpoints** are created dynamically at various points during the execution.
  - □ **If a fault is detected, the system returns to the last checkpoint and continues the execution.**
  - □ **Fault detection checks need to be embedded in the code and executed before the checkpoints are created.**

Checkpoint and Restart

- **A number of factors influence the efficiency of checkpointing, including:**
  - □ execution requirements,
  - □ the interval between checkpoints,
  - □ fault activation rate and
  - □ overhead associated with creating fault detection checks, checkpoints, recovery, etc.

Checkpoint and Restart

- **In static approach**, the expected time to complete the execution grows exponentially with the execution requirements.
  - □ Therefore, static checkpointing is effective only if the processing requirement is relatively small.
- **In dynamic approach**, it is possible to achieve linear increase in execution time as the processing requirements grow.

Checkpoint and Restart

- **There are three strategies for dynamic placing of checkpoints:**
  - □ *Equidistant (دارای مسافت مساوی)*, which places checkpoints at deterministic fixed time intervals. The time between checkpoints is chosen depending on the expected fault rate.
  - □ *Modular*, which places checkpoints at the end of the sub-modules in a module, after the fault detection checks for the sub-module are completed. The execution time depends on the distribution of the sub-modules and expected fault rate.
  - □ *Random*, placing checkpoints at random.

Checkpoint and Restart

- **Overall, restart recovery mechanism has the following advantages:**
  - □ It is conceptually simple.
  - □ It is independent of the damage caused by a fault.
  - □ It is applicable to unanticipated faults.
  - □ It is general enough to be used at multiple levels in a system.

Checkpoint and Restart

- **A problem with restart recovery is that *non-recoverable actions* exist in some systems.**
- These actions are usually associated with external events that cannot be compensated by simply reloading the state and restarting the system.
  - □ Examples of non-recoverable actions are **firing a missile** or **soldering (جوش دادن) a pair of wires**.
- The recovery from such actions need to include special treatment, for example by compensating for their consequences (e.g. undoing a solder), or delaying their output until after additional confirmation checks are completed (e.g. do a friend-or-foe (دوست یا دشمن) confirmation before firing).

Process Pairs

- Process pair technique runs two identical versions of the software on separate processors (Figure 7.2).

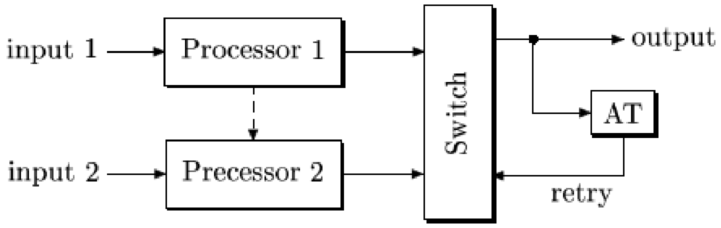
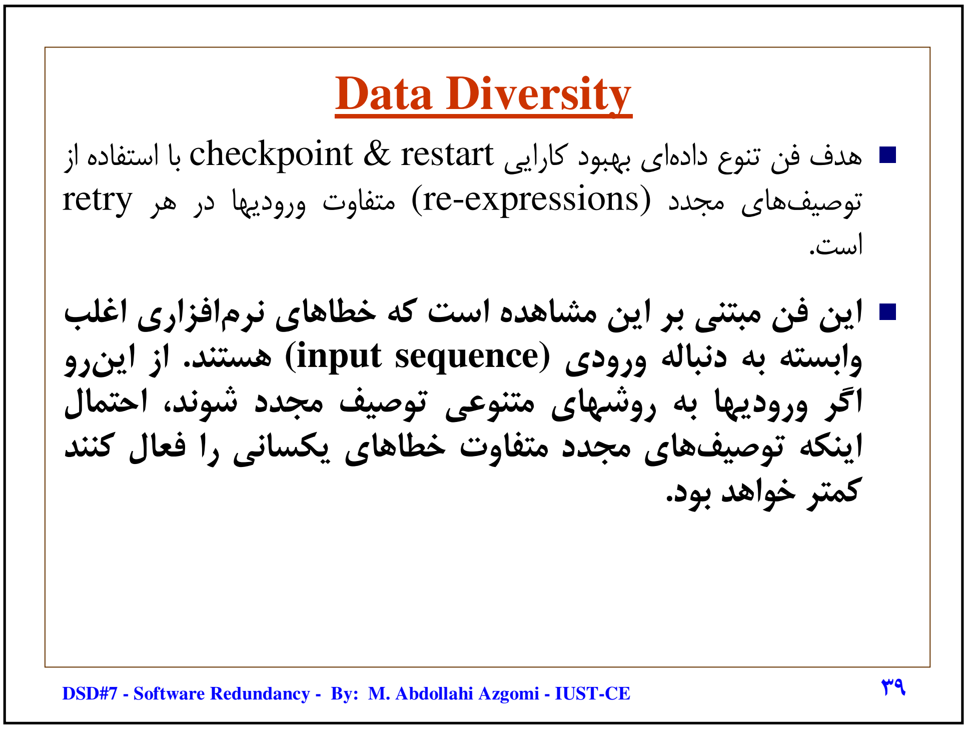


Figure 7.2. Process pairs.

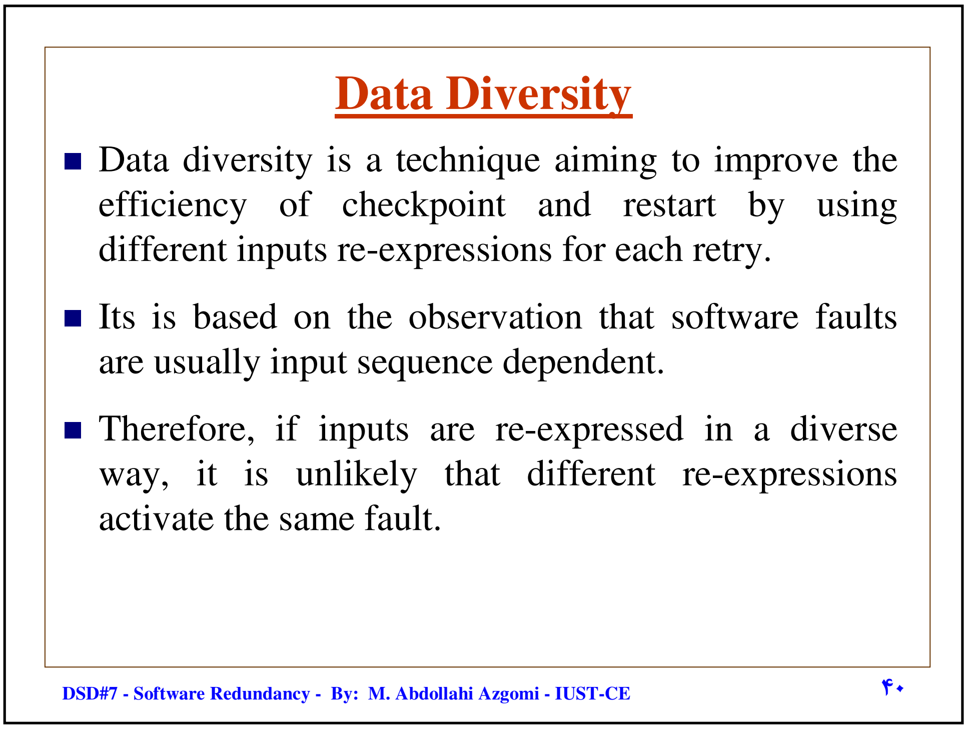
Process Pairs

- First the primary processor, *Processor* 1, is active. It executes the program and sends the checkpoint information to the secondary processor, *Processor* 2. If a fault is detected, the primary processor is switched off.
- The secondary processor loads the last checkpoint as its starting state and continues the execution. The *Processor* 1 executes diagnostic checks off-line. If the fault is non-recoverable, the replacement is performed. After returning to service, the repaired processor becomes secondary processor.



Data Diversity

- هدف فن تنوع داده‌ای بهبود کارایی checkpoint & restart با استفاده از توصیف‌های مجدد (re-expressions) متفاوت ورودیها در هر retry است.
- **این فن مبتنی بر این مشاهده است که خطاهای نرم‌افزاری اغلب وابسته به دنباله ورودی (input sequence) هستند. از این‌رو اگر ورودیها به روشهای متنوعی توصیف مجدد شوند، احتمال اینکه توصیف‌های مجدد متفاوت خطاهای یکسانی را فعال کنند کمتر خواهد بود.**



Data Diversity

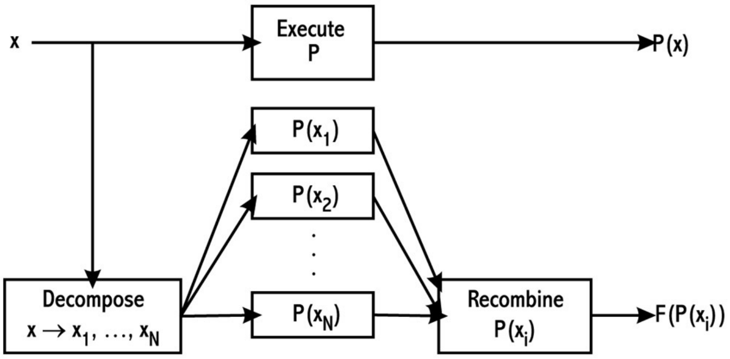
- Data diversity is a technique aiming to improve the efficiency of checkpoint and restart by using different inputs re-expressions for each retry.
- Its is based on the observation that software faults are usually input sequence dependent.
- Therefore, if inputs are re-expressed in a diverse way, it is unlikely that different re-expressions activate the same fault.

Data Diversity

- **There are three basic techniques for data diversity:**
  - □ Input data **re-expression**, where only the input is **changed**.
    - مثل گرد کردن (توصیف مجدد) ورودیهای خوانده شده از سنسور
  - □ Input data **re-expression** with **post-execution** adjustment, where the output result also needs to be adjusted in accordance with a given set of rules.
    - For example, if the inputs were re-expressed by **encoding** them in some code, then the output result is decoded following the **decoding** rules of the code.
    - مثل تبدیل حروف فارسی یک رشته به حروف زیر ۱۲۸ (توصیف مجدد با رمزگذاری) و معکوس کردن رشته و سپس تبدیل بازیابی دوباره حروف فارسی (رمزگشایی).
  - □ Input data re-expression via decomposition and re-combination, where the input is decomposed into smaller parts and then re-combined after execution to obtain the output result.
    - مثل تقسیم یک رشته به چند زیررشته (توصیف مجدد)، بعد اجرای یک تابع روی زیررشته‌ها و سپس الحاق زیررشته‌ها (ترکیب مجدد).

(خارج از کتاب)Data Diversity

- Using data re-expression algorithms (DRA) to obtain logically equivalent variants of the input data



Data re-expression via decomposition and recombination

(خارج از کتاب)Data Diversity

- This technique might not be acceptable to all programs since equivalent input data transformations might not be acceptable by the specification.
- However, in some cases like a real time control program, a minor perturbation in sensor values may be able to prevent a failure since sensor values are usually noisy and inaccurate .

Data Diversity

- Data diversity can also be used in combination with the multi-version fault tolerance techniques, presented in the next section.

3. Multi-Version Techniques

- Multi-version techniques use two or more versions of the same software module, which satisfy the **design diversity** (تنوع طراحی) requirements.
  - □ For example, different teams, different coding languages or different algorithms can be used to maximize the probability that all the versions do not have common faults.

Recovery Blocks

- The **recovery blocks** (بلوکهای بازیافتی) technique combines checkpoint and restart approach with **standby sparing redundancy** scheme.
- The basic configuration is shown in Figure 7.3. …

Recovery Blocks

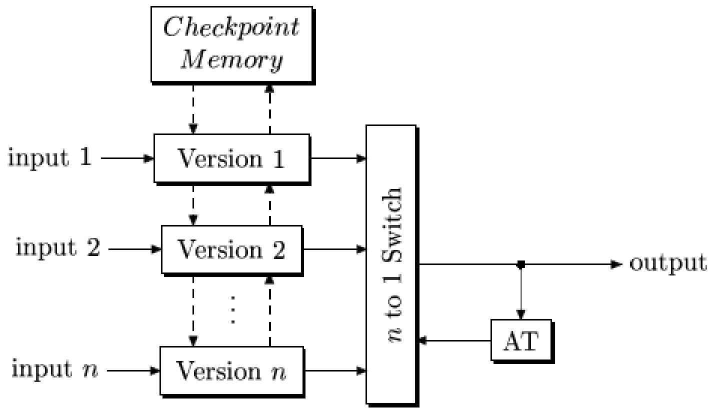


Figure 7.3. Recovery blocks.

Recovery Blocks

- Versions 1 to *n* represent **different implementations** of the same program.
- Only one of the versions provides the system's output.
- If an error if detected by the acceptance test, a retry signal is sent to the switch.
- The system is rolled back to the state stored in the checkpoint memory and the switch then switches the execution to another version of the module.

Recovery Blocks

- Checkpoints are created before a version executes.
- Various checks are used for acceptance testing of the active version of the module.
- The check should be kept simple in order to maintain execution speed.
- Check can either be placed at the output for a module, or embedded in the code to increase the effectiveness of fault detection.

Recovery Blocks

- Similarly to **cold and hot** versions of hardware standby sparing technique, different versions can be executed either **serially**, or **concurrently**, depending on available processing capability and performance requirements.
  - □ **Serial execution** may require the use of checkpoints to reload the state before the next version is executed. The cost in time of trying multiple versions serially may be too expensive, especially for a real-time system.
  - □ However, a **concurrent system** requires the expense of *n* redundant hardware modules, a communications network to connect them and the use of **input and state consistency algorithms** (?!?).

Recovery Blocks

- If all *n* versions are tried and failed, the module invokes the **exception handler** to communicate to the rest of the system a failure to complete its function.

Recovery Blocks معایب

- As all multi-version techniques, recovery blocks technique is heavily dependent on design diversity.
- The recovery blocks method increases the pressure on the specification to be detailed enough to create different multiple alternatives that are functionally the **same**.
  - □ *This issue is further discussed in Section 3.4.*
- In addition, acceptance tests suffer from lack of guideness for their development.
  - □ They are **highly application dependent**, they are difficult to create and they cannot test for a specific correct answer, but only for “acceptable” values.

N-Version Programming

- The **N-version programming (NVP)** techniques resembles the *N*-modular hardware redundancy. The block diagram is shown in Figure 7.4.

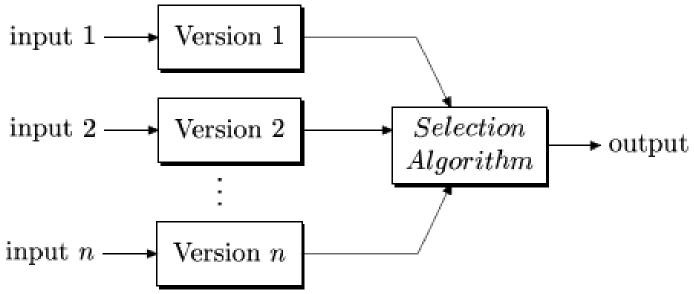


Figure 7.4. *N*-version programming.

N-Version Programming

- **It consists of *n* different software implementations of a module, executed concurrently.**
- Each version accomplishes the same task, but in a different way.
- The **selection algorithm** decides which of the answers is correct and returns this answer as a result of the modules execution.
- The selection algorithm is usually implemented as a **generic voter**.
- This is an advantage over recovery block fault detection mechanism, **requiring application dependent acceptance tests**.

N-Version Programming

- Many different types of voters has been developed, including
  - □ **formalized majority voter (رای گیری اکثریت)**,
  - □ **generalized median voter (رای گیری میانه)**,
  - □ **formalized plurality voter (رای گیری تعدد)** and
  - □ **weighted averaging technique (فن میانگین گیری وزن دار)**.

N-Version Programming

- The voters have the capability to perform inexact voting by using the concept of *metric space* (X, d).
  - □ The set X is the output space of the software and d is a metric function that associates any two elements in X with a real-valued number.
- **Definition of metric:** A ***metric*** is a function that associates any two objects in a set with a number and that preserves a number of properties of the distance with which we are familiar.

N-Version Programming

- The inexact values are declared equal if their metric distance is less than some pre-defined threshold *e*.
- In the ***formalized majority voter***, the outputs are compared and, if more than half of the values agree, the voter output is selected as one of the values in the agreement group.
- The ***generalized median voter*** selects the median of the values as the correct result.
  - □ The median is computed by successively eliminating pair of values that are farther (دورتر) apart until only one value remains.

N-Version Programming

- The ***formalized plurality voter*** partitions the set of outputs based on metric equality and selects the output from the largest partition group.
- The ***weighted averaging technique*** combines the outputs in a weighted average to produce the result.
  - □ The weight can be selected in advance based on the characteristics of the individual versions.
  - □ If all the weights are equal, this technique reduces to the median selection technique.

N-Version Programming

- The **selection algorithms** are normally developed taking into account the consequences of erroneous output for dependability attributes like reliability, availability and safety.
  - □ **For applications where reliability is important, the selection algorithm should be designed so that the selected result is correct with a very high probability.**
  - □ **If availability is an issue, the selection algorithm is expected to produce an output even if it is incorrect.**
    - Such an approach would be acceptable as long as the program execution in not subsequently dependent on previously generated (possibly erroneous) results.
  - □ **For applications where safety is the main concern, the selection algorithm is required to correctly distinguish the erroneous version and mask its results.**
    - In cases when the algorithm cannot select the correct result with a high confidence, it should report to the system an error condition or initiate an acceptable safe output sequence.

N-Version Programming

- ***N*-version programming technique can tolerate the design faults present in the software if the design diversity concept is implemented properly.**
- Each version of the module should be implemented in an as diverse as possible manner, including
  - □ different tool sets,
  - □ different programming languages, and
  - □ possibly different environments.

N-Version Programming

- The various development groups must have as little interaction related to the programming between them as possible.
- The specification of the system is required to be detailed enough so that the various versions are completely compatible.
- On the other hand, the specification should be flexible to give the programmer a possibility to create diverse designs.

N Self-Checking Programming

- ***N* self-checking programming** combines recovery blocks concept with *N*-version programming.
- The **checking** is performed either by using **acceptance tests**, or by using **comparison**.
  - □ Examples of applications of *N* self-checking programming are Lucent ESS-5 phone switch and the Airbus A-340 airplane.

N Self-Checking Programming

- *N* self-checking programming using acceptance tests is shown in Figure 7.5.

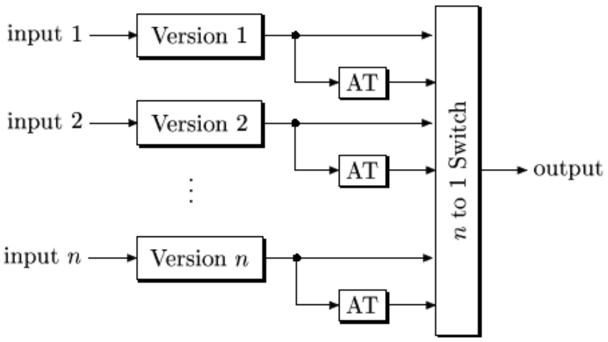


Figure 7.5. *N* self-checking programming using acceptance tests.

N Self-Checking Programming

- Different versions of the program module and the acceptance tests AT are developed independently from common requirements.
  - □ The individual checks for each of the version are either embedded in the code, or placed at the output.
- The use of separate acceptance tests for each version is the main difference of this technique from recovery blocks approach.

N Self-Checking Programming

- The execution of each version can be done either serially, or concurrently.
- In both cases, the output is taken from the highest-ranking version which passes its acceptance test.

N Self-Checking Programming

- *N* self-checking programming using comparison is shown in Figure 7.6.

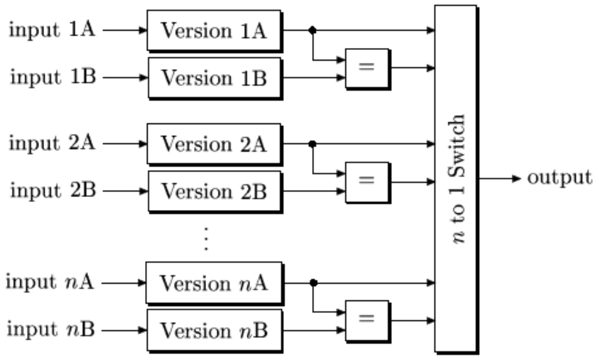


Figure 7.6. *N* self-checking programming using comparison.

N Self-Checking Programming

- The scheme resembles triplex-duplex hardware redundancy.
- An advantage over *N* self-checking programming using acceptance tests is that an application **independent decision algorithm (*comparison*)** is used for fault detection.

Design Diversity

- The most critical issue in multi-version software fault tolerance techniques is assuring independence between the different versions of software through design diversity (تنوع طراحی).
- **Design diversity aims to protect the software from containing common design faults.**
- Software systems are vulnerable to common design faults if **they are developed by the same design team, by applying the same design rules and using the same software tools.**

Design Diversity

- Presently, the implementation of design diversity remains a controversial (بحث‌انگیز) subject.
- The increase in complexity caused by redundant multiple versions can be quite severe (سخت) and may result in a less dependent system (سیستم کمتر اتکاءپذیر), unless appropriate measures are taken (مگر آنکه ابزارهای مناسب استفاده شوند).

Design Diversity

- **Decision to be made when developing a multi-version software system include:**
  - □ which modules are to be made redundant (usually less reliable modules are chosen);
  - □ the level of redundancy (procedure, process, whole system);
  - □ the required number of redundant versions;
  - □ the required diversity (diverse specification, algorithm, code, programming language, testing technique, etc.);
  - □ rules of isolation between the development teams, to prevent the flow of information that could result in common design error.

Design Diversity

- The cost of development of a multi-version software also needs to be taken into account.
- A direct replication of the full development effort would have a total cost prohibitive for most applications.
- The cost can be reduced by allocating redundancy to dependability critical parts of the system only.
- When the cost of alternative dependability improvement techniques is high because of the need for specialized stuff and tools, the use of design diversity can result in cost savings.

4. Software Testing

- **Software testing is the process of executing a program with the intent of finding errors** [Beizer, 1990].
- Testing is a major consideration in software development.
- In many organizations, more time is devoted to testing than to any other phase of software development.
- On complex projects, test developers might be twice or three times as many as code developers on a project team.

4. Software Testing

- **There are two types of software testing: functional and structural.**
  - □ *Functional testing* (also called *behavioral testing*, *black-box testing*, *closed-box testing*), compares test program behavior against its specification.
  - □ *Structural testing* (also called white-box testing, glass-box testing) checks the internal structure of a program for errors.

4. Software Testing

- **For example, suppose we test a program which adds two integers.**
  - □ The goal of functional testing is to verify whether the implemented operation is indeed addition instead of e.g. multiplication.
  - □ Structural testing does not question the functionally of the program, but checks whether the internal structure is *consistent…*

- با تست کردن همه مسیرهای اجرا ... مفهوم پوشش (coverage)
که در ادامه تشریح می‌شود.

4. Software Testing

- **A strength of the structural approach is that the entire software implementation is taken into account during testing, which facilitates error detection even when the software specification is vague (مبهم) or incomplete.**

4. Software Testing

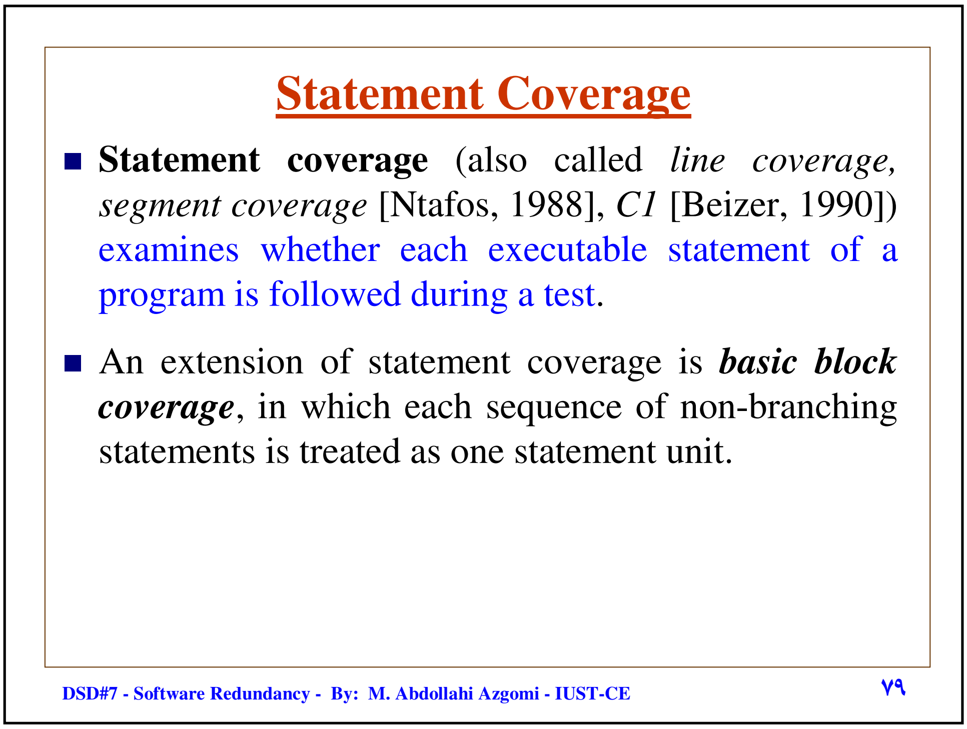
- The effectiveness of structural testing is normally expressed in terms of **test coverage metrics**, which measure the fraction of code exercised by test cases.
- **Common test coverage metrics are** [Beizer, 1990]:
  - □ **statement coverage**,
  - □ **branch coverage**, and
  - □ **path coverage**.

4. Software Testing

- *Statement* coverage requires that the program under test is run with enough test cases, so that all its statements are executed at least once.
- *Decision* coverage requires that all branches of the program are executed at least once.
- *Path* coverage requires that each of the possible paths through the program is followed.
  - □ Path coverage is the most reliable metric, however, it is not applicable to large systems, since the number of paths is exponential to the number of branches.

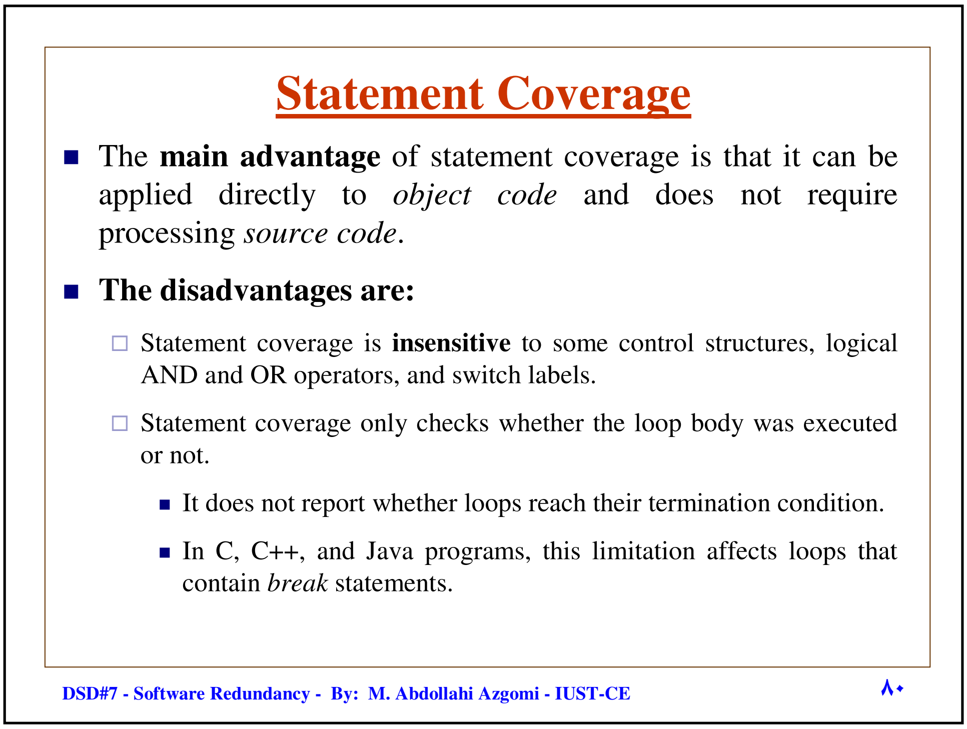
4. Software Testing

- This section describes a technique for structural testing which finds a part of program's flowgraph, called *kernel*, with the property that any set of tests which executes all *vertices* (edges) of the kernel executes all *vertices* (edges) of the flowgraph [Dubrova, 2005].



Statement Coverage

- **Statement coverage** (also called *line coverage, segment coverage* [Ntafos, 1988], *C1* [Beizer, 1990]) examines whether each executable statement of a program is followed during a test.
- An extension of statement coverage is ***basic block coverage***, in which each sequence of non-branching statements is treated as one statement unit.

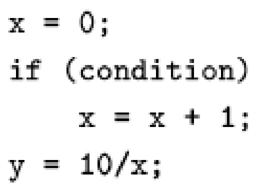


Statement Coverage

- The **main advantage** of statement coverage is that it can be applied directly to *object code* and does not require processing *source code*.
- **The disadvantages are:**
  - □ Statement coverage is **insensitive** to some control structures, logical AND and OR operators, and switch labels.
  - □ Statement coverage only checks whether the loop body was executed or not.
    - It does not report whether loops reach their termination condition.
    - In C, C++, and Java programs, this limitation affects loops that contain *break* statements.

Statement Coverage

- As an example of the insensitivity of statement coverage to some control structures, consider the following code:



```
x = 0;
if (condition)
     x = x + 1;
y = 10/x;
```

Statement Coverage

- If there is no test case which causes **`condition`** to evaluate false, the error in this code will not be detected in spite of 100% statement coverage.
- The error will appear only if **`condition`** evaluates false for some test case.
- Since **`if`**-statements are common in programs, this problem is a serious drawback of statement coverage.

Branch Coverage

- **Branch coverage** (also referred to as *decision coverage, all-edges coverage* [Roper, 1994], *C2* [Beizer, 1990]) requires that each branch of a program is executed at least once during a test.
- Boolean expressions of `if`- or `while`-statements are checked to be evaluated to both *true* and *false*.
- The entire Boolean expression is treated as one predicate regardless of whether it contains logical AND and OR operators.
- `switch` statements, exception handlers, and interrupt handlers are treated similarly.
- Decision coverage includes statement coverage since executing every branch leads to executing every statement.

Branch Coverage

- **An advantage of branch coverage is its relative simplicity.**
  - □ It allows overcoming many problems of statement coverage.
- However, it might miss some errors as demonstrated by the following example:

```
if (condition1)
    x = 0;
else
    x = 2;
if (condition2)
    y = 10*x;
else
    y = 10/x;
```

Branch Coverage

- The 100% branch coverage can be achieved by two test cases which cause both **`condition1`** and **`condition2`** to evaluate *true*, and both **`condition1`** and **`condition2`** to evaluate *false*.
- However, the error which occurs when **`condition1`** evaluates *true* and **`condition2`** evaluates *false* will not be detected by these two tests.

Path Coverage

- The error in the example above can be detected by exercising every **path** through the program.
- However, since the number of paths is exponential to the number of branches, testing every path is not possible for large systems.
- For example, if one test case takes 0.1×10^{-5} seconds to execute, then testing all paths of a program containing 30 **`if`**-statements will take 18 minutes and testing all paths of a program with 60 **`if`**-statements will take **366 centuries** (?!?!).