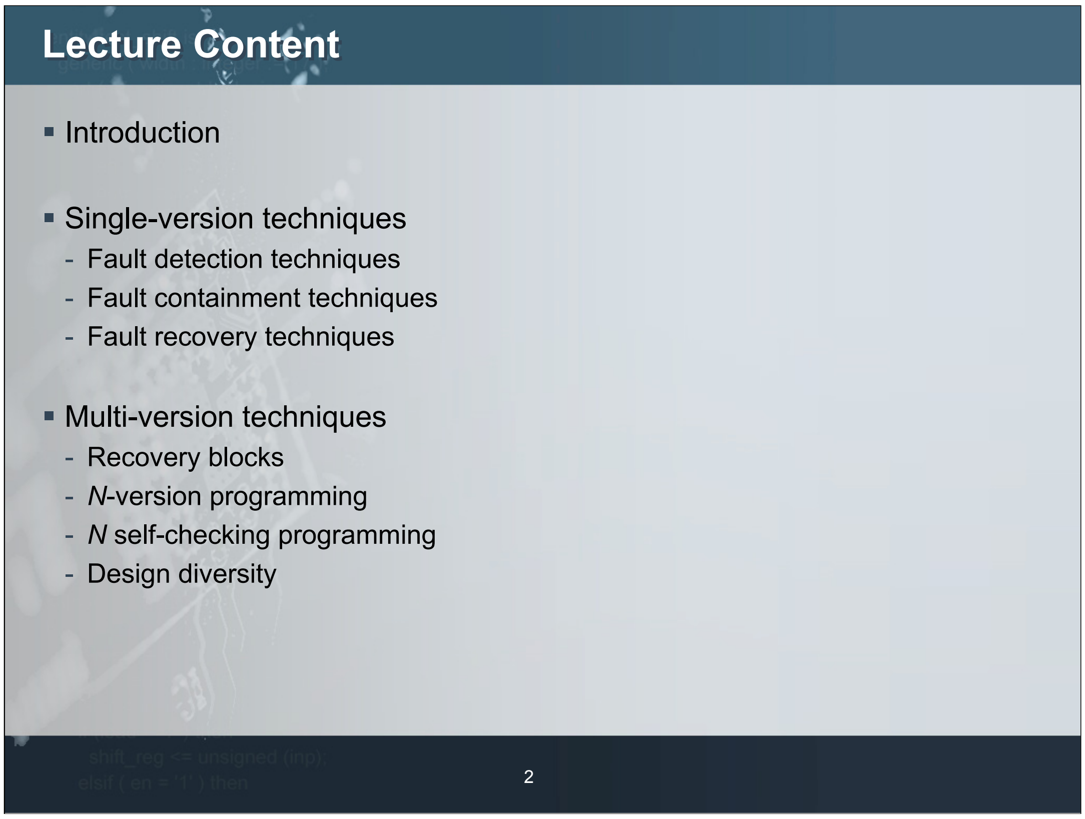
Digitalni sistemi otporni na otkaz

Predavanje X



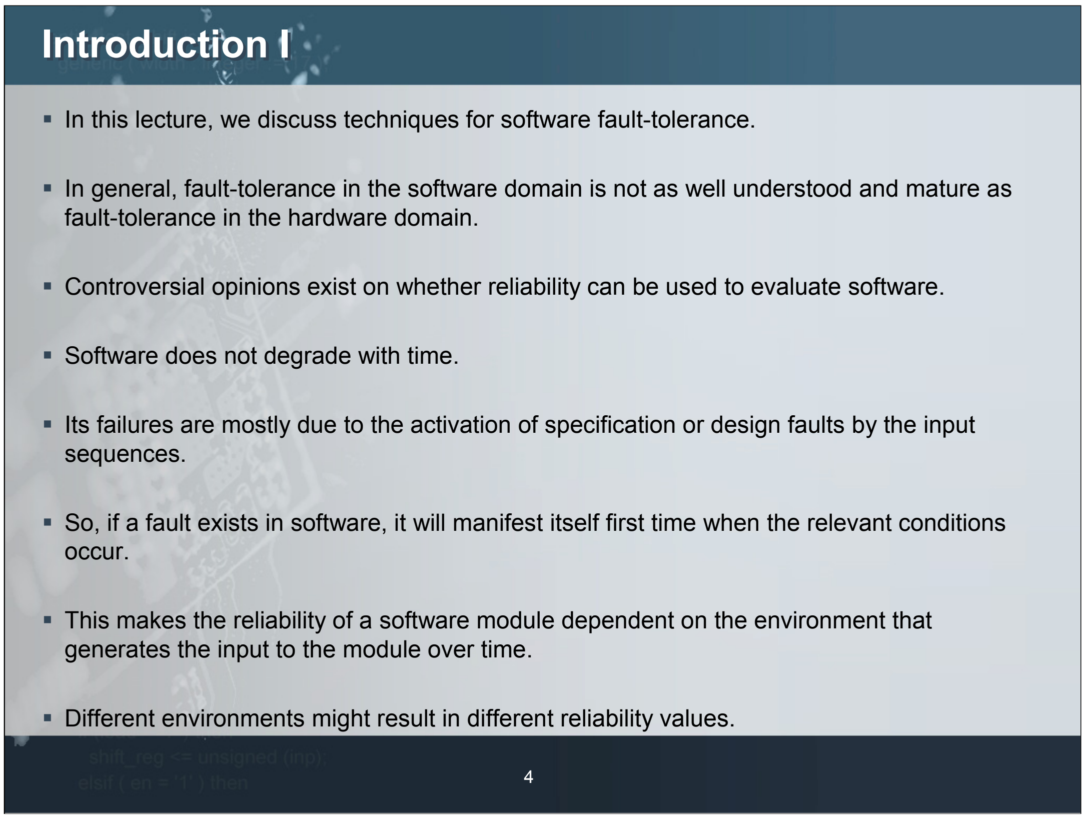
Lecture Content

- Introduction
- Single-version techniques
  - Fault detection techniques
  - Fault containment techniques
  - Fault recovery techniques
- Multi-version techniques
  - Recovery blocks
  - *N*-version programming
  - *N* self-checking programming
  - Design diversity

2

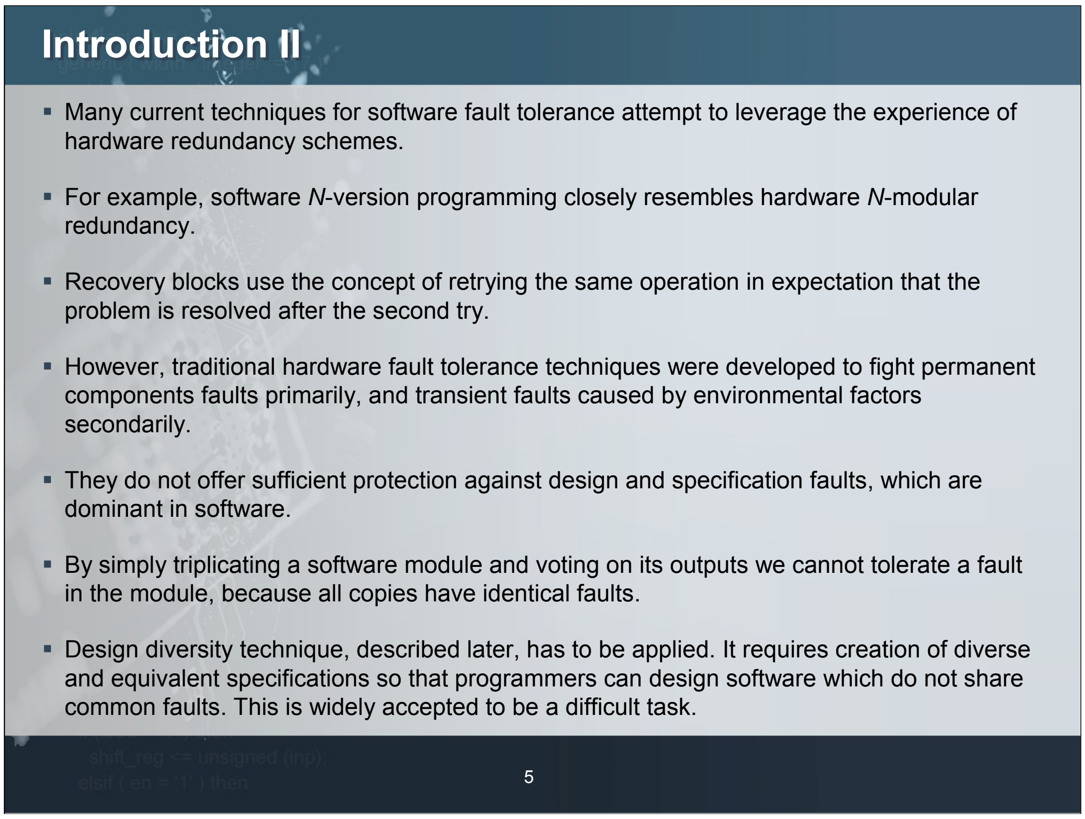
Software Fault Tolerance Techniques

Software Fault Tolerance



Introduction I

- In this lecture, we discuss techniques for software fault-tolerance.
- In general, fault-tolerance in the software domain is not as well understood and mature as fault-tolerance in the hardware domain.
- Controversial opinions exist on whether reliability can be used to evaluate software.
- Software does not degrade with time.
- Its failures are mostly due to the activation of specification or design faults by the input sequences.
- So, if a fault exists in software, it will manifest itself first time when the relevant conditions occur.
- This makes the reliability of a software module dependent on the environment that generates the input to the module over time.
- Different environments might result in different reliability values.



Introduction II

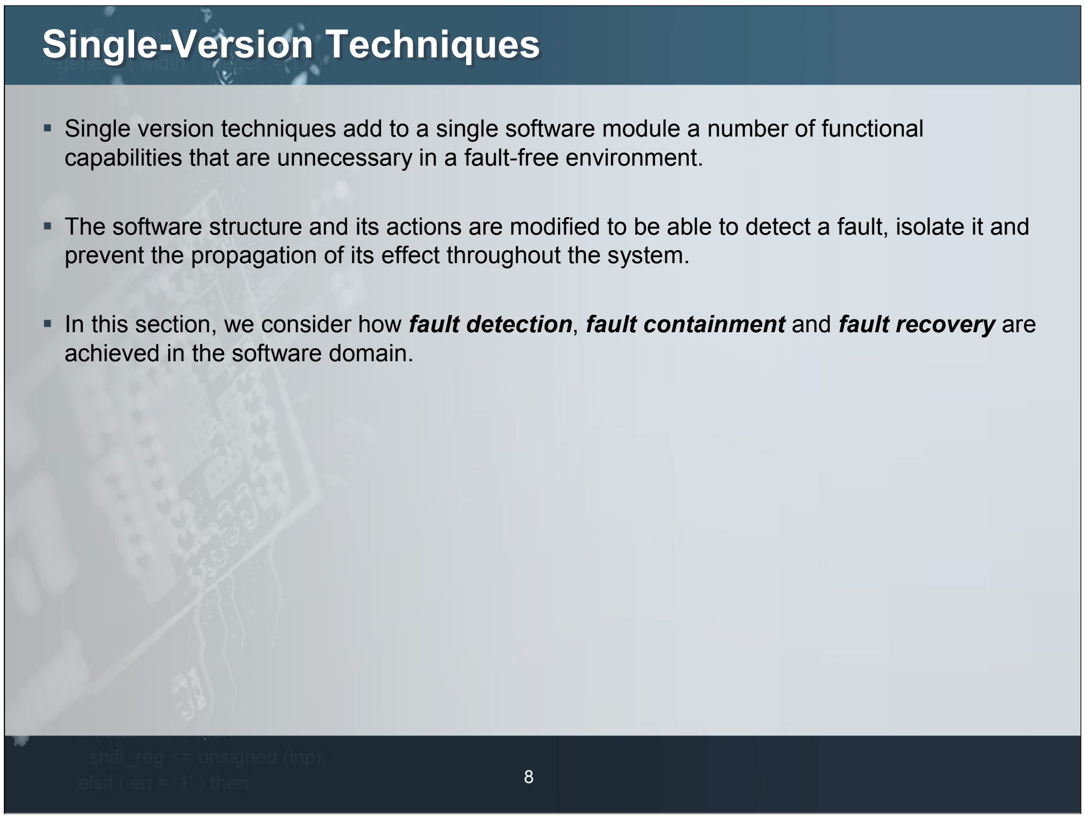
- Many current techniques for software fault tolerance attempt to leverage the experience of hardware redundancy schemes.
- For example, software *N*-version programming closely resembles hardware *N*-modular redundancy.
- Recovery blocks use the concept of retrying the same operation in expectation that the problem is resolved after the second try.
- However, traditional hardware fault tolerance techniques were developed to fight permanent components faults primarily, and transient faults caused by environmental factors secondarily.
- They do not offer sufficient protection against design and specification faults, which are dominant in software.
- By simply triplicating a software module and voting on its outputs we cannot tolerate a fault in the module, because all copies have identical faults.
- Design diversity technique, described later, has to be applied. It requires creation of diverse and equivalent specifications so that programmers can design software which do not share common faults. This is widely accepted to be a difficult task.

Introduction III

- 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.
- As in the hardware case, a number of possibilities has to be examined to determine at which level the redundancy needs to be provided and which modules are to be made redundant.
- The redundancy can be applied to a procedure, or to a process, or to the whole software system.
- Usually, the components which have high probability of faults are chosen to be made redundant.
- As in the hardware case, the increase in complexity caused by the redundancy can be quite severe and may diminish the dependability improvement, unless redundant resources are allocated in a proper way.

Software Fault Tolerance Techniques

Single-Version Techniques



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.
- The software structure and its 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 the software domain.

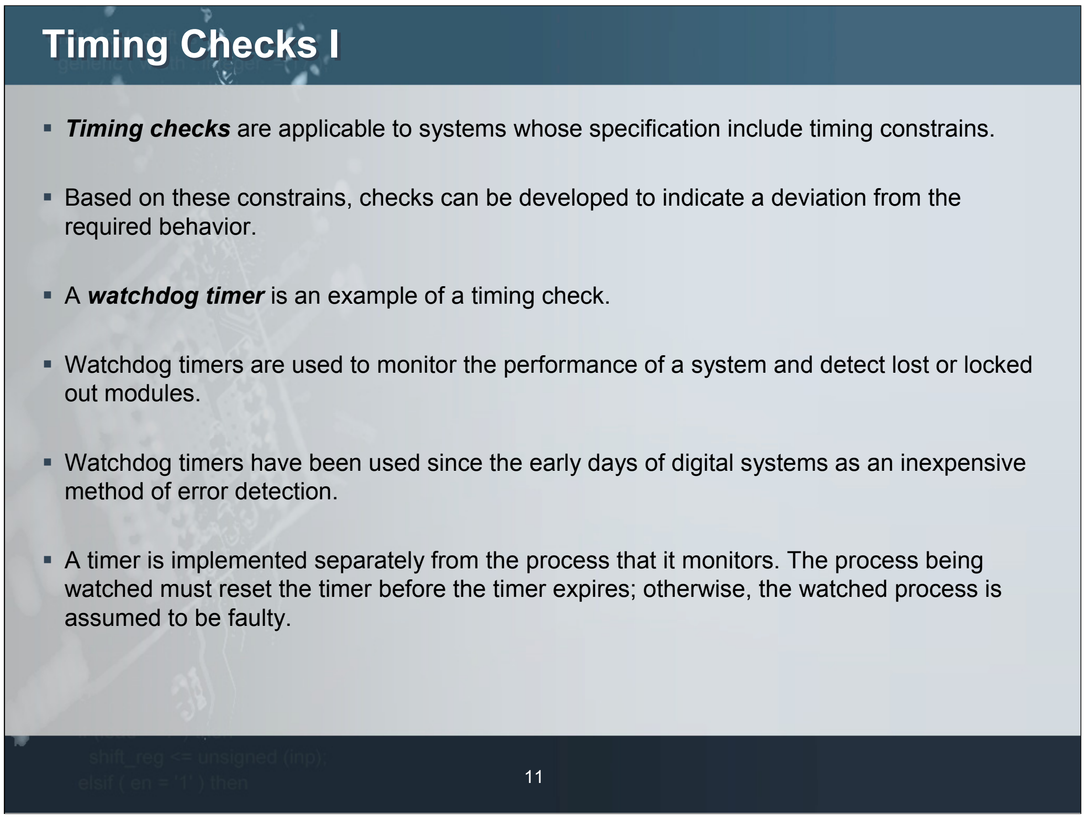
8

Software Fault Tolerance Techniques

Fault Detection Techniques

Fault Detection Techniques

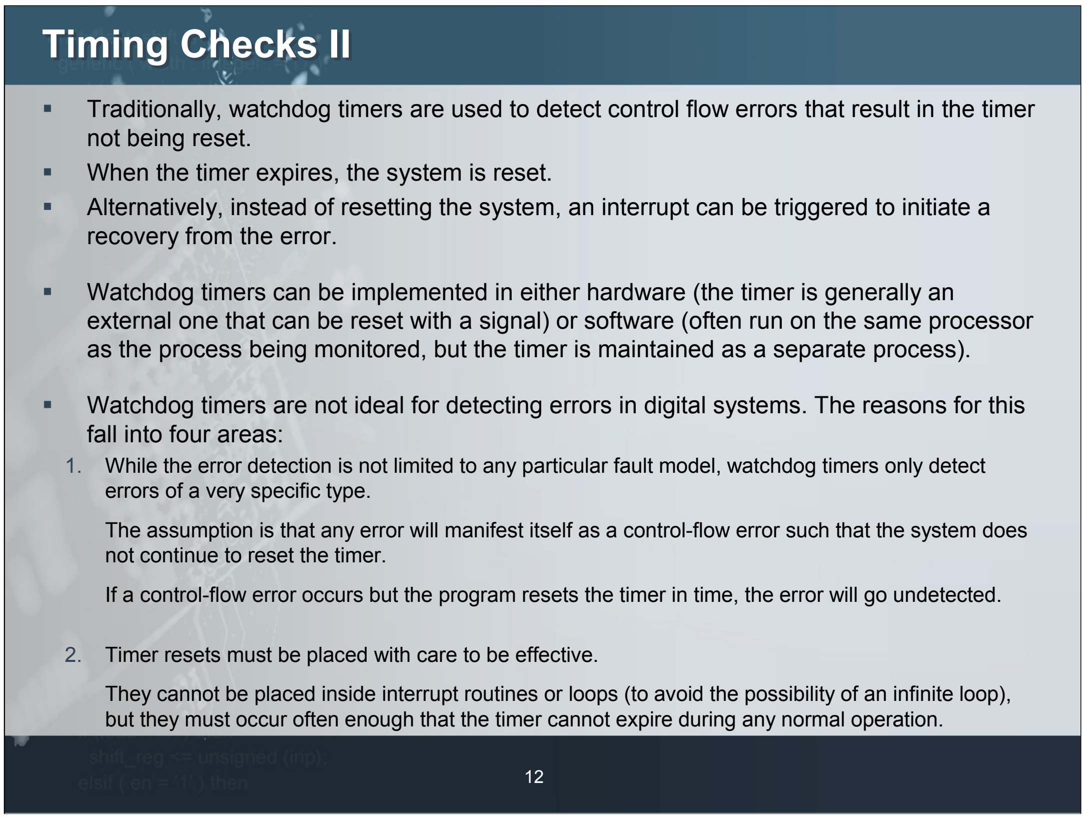
- As in the hardware case, the goal of fault detection in software is to determine that a fault has occurred within a system.
- Single-version fault tolerance techniques usually use various types of ***acceptance tests*** to detect faults.
- The result of a program is subjected to a test.
- If the result passes the test, the program continues its execution.
- A failed test indicates a fault.
- A test is most effective if it can be calculated in a simple way and if it is based on criteria that can be derived independently of the program application.
- The existing techniques include ***timing checks***, ***coding checks***, ***reversal checks***, ***reasonableness checks*** and ***structural checks***.



Timing Checks I

- ***Timing checks*** are applicable to systems whose specification include timing constrains.
- Based on these constrains, checks can be developed to indicate a deviation from the required behavior.
- A ***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.
- Watchdog timers have been used since the early days of digital systems as an inexpensive method of error detection.
- A timer is implemented separately from the process that it monitors. The process being watched must reset the timer before the timer expires; otherwise, the watched process is assumed to be faulty.

11



Timing Checks II

- Traditionally, watchdog timers are used to detect control flow errors that result in the timer not being reset.
- When the timer expires, the system is reset.
- Alternatively, instead of resetting the system, an interrupt can be triggered to initiate a recovery from the error.

- Watchdog timers can be implemented in either hardware (the timer is generally an external one that can be reset with a signal) or software (often run on the same processor as the process being monitored, but the timer is maintained as a separate process).

- Watchdog timers are not ideal for detecting errors in digital systems. The reasons for this fall into four areas:
  1. While the error detection is not limited to any particular fault model, watchdog timers only detect errors of a very specific type.

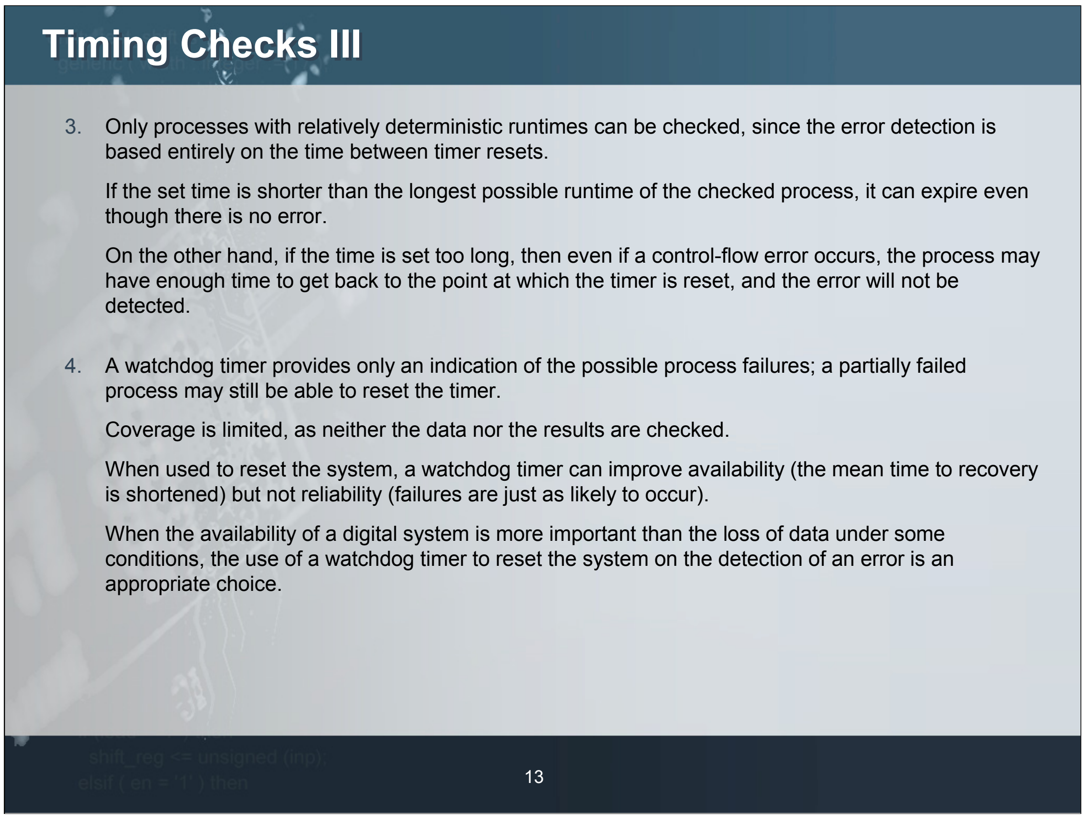
     The assumption is that any error will manifest itself as a control-flow error such that the system does not continue to reset the timer.

     If a control-flow error occurs but the program resets the timer in time, the error will go undetected.

  2. Timer resets must be placed with care to be effective.

     They cannot be placed inside interrupt routines or loops (to avoid the possibility of an infinite loop), but they must occur often enough that the timer cannot expire during any normal operation.

12



Timing Checks III

3. Only processes with relatively deterministic runtimes can be checked, since the error detection is based entirely on the time between timer resets.

   If the set time is shorter than the longest possible runtime of the checked process, it can expire even though there is no error.

   On the other hand, if the time is set too long, then even if a control-flow error occurs, the process may have enough time to get back to the point at which the timer is reset, and the error will not be detected.

4. A watchdog timer provides only an indication of the possible process failures; a partially failed process may still be able to reset the timer.

   Coverage is limited, as neither the data nor the results are checked.

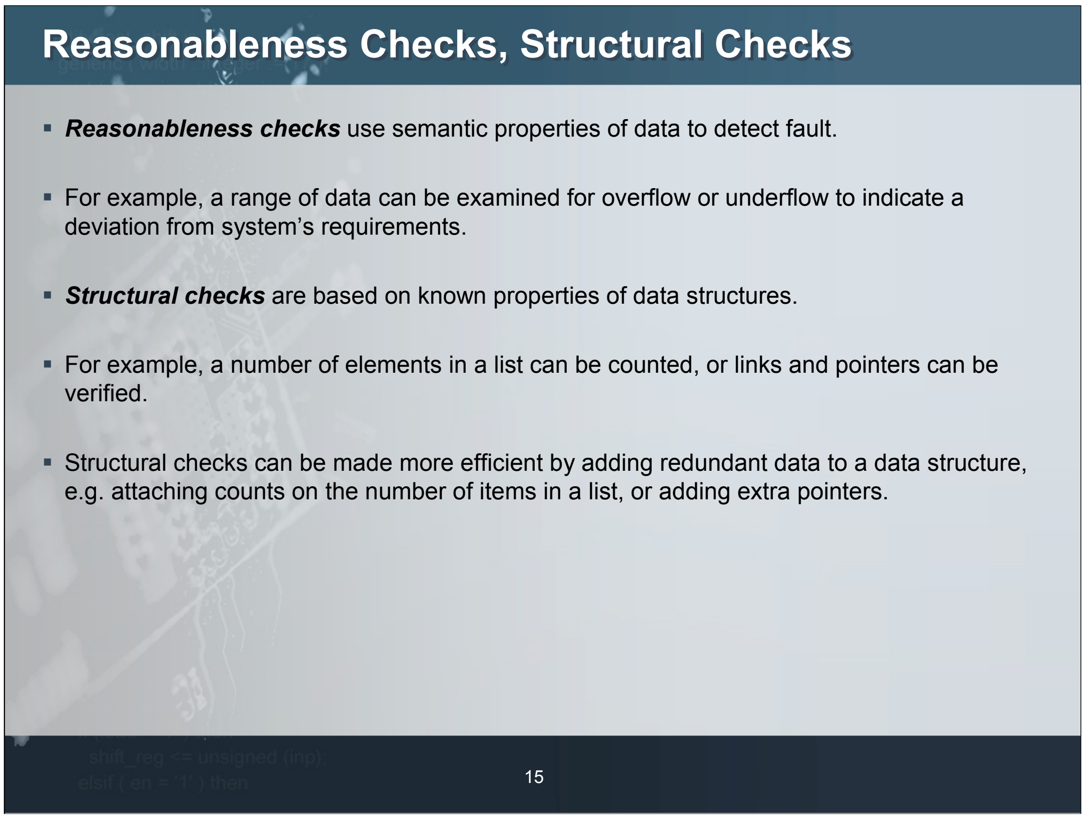
   When used to reset the system, a watchdog timer can improve availability (the mean time to recovery is shortened) but not reliability (failures are just as likely to occur).

   When the availability of a digital system is more important than the loss of data under some conditions, the use of a watchdog timer to reset the system on the detection of an error is an appropriate choice.

13

Coding Checks, Reversal Checks

- ***Coding checks*** are applicable to systems whose data can be encoded using information redundancy techniques.
- Cyclic redundancy checks 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.
- 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.



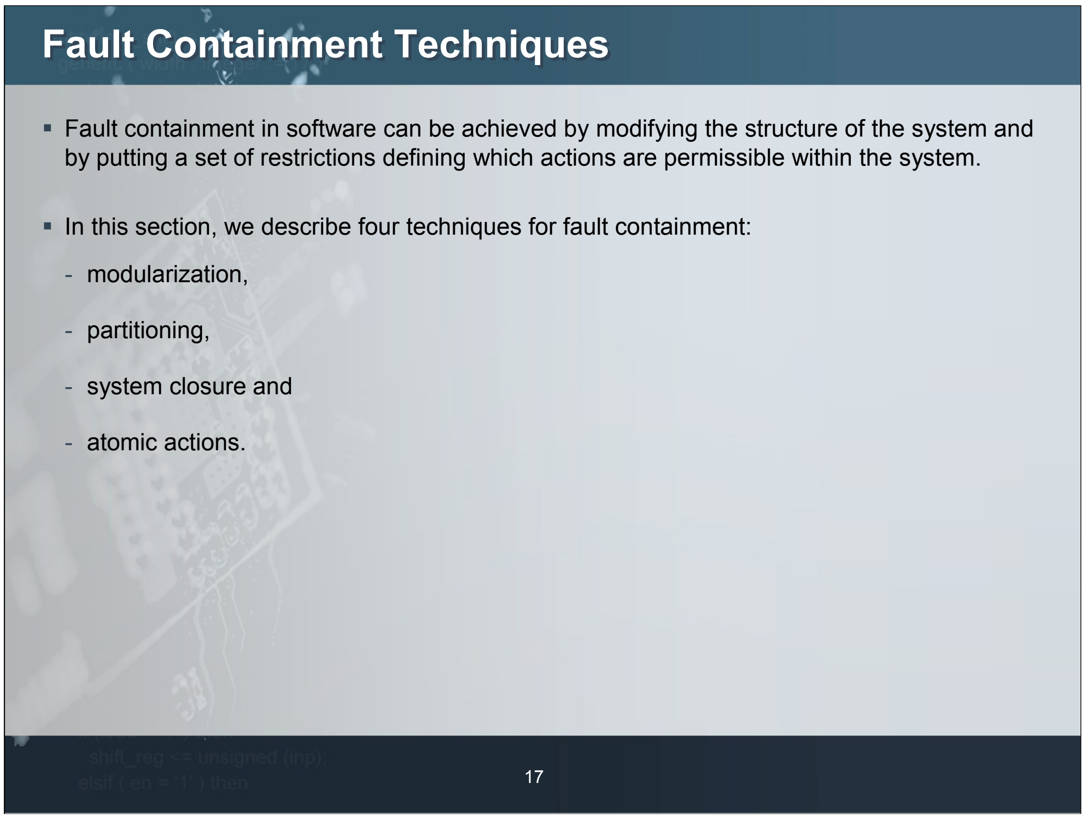
Reasonableness Checks, Structural Checks

- ***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.
- ***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.

15

Software Fault Tolerance Techniques

Fault Containment Techniques



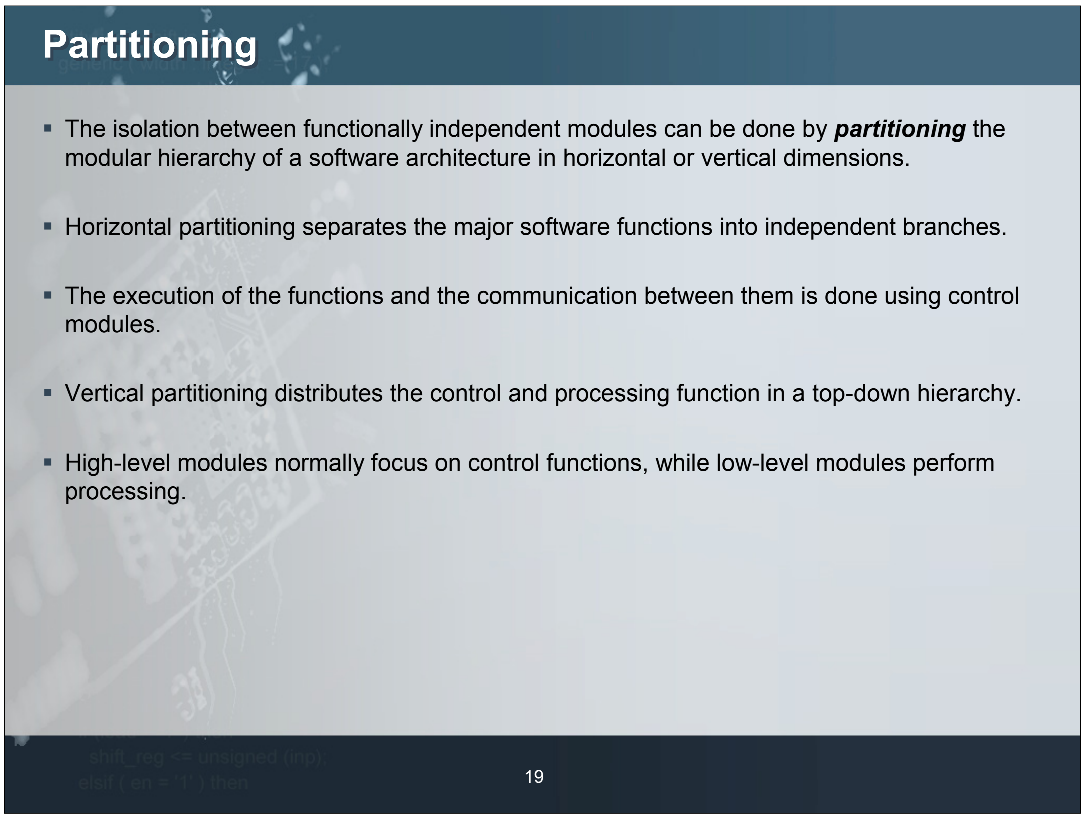
Fault Containment Techniques

- Fault containment 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.
- In this section, we describe four techniques for fault containment:
  - modularization,
  - partitioning,
  - system closure and
  - atomic actions.

17

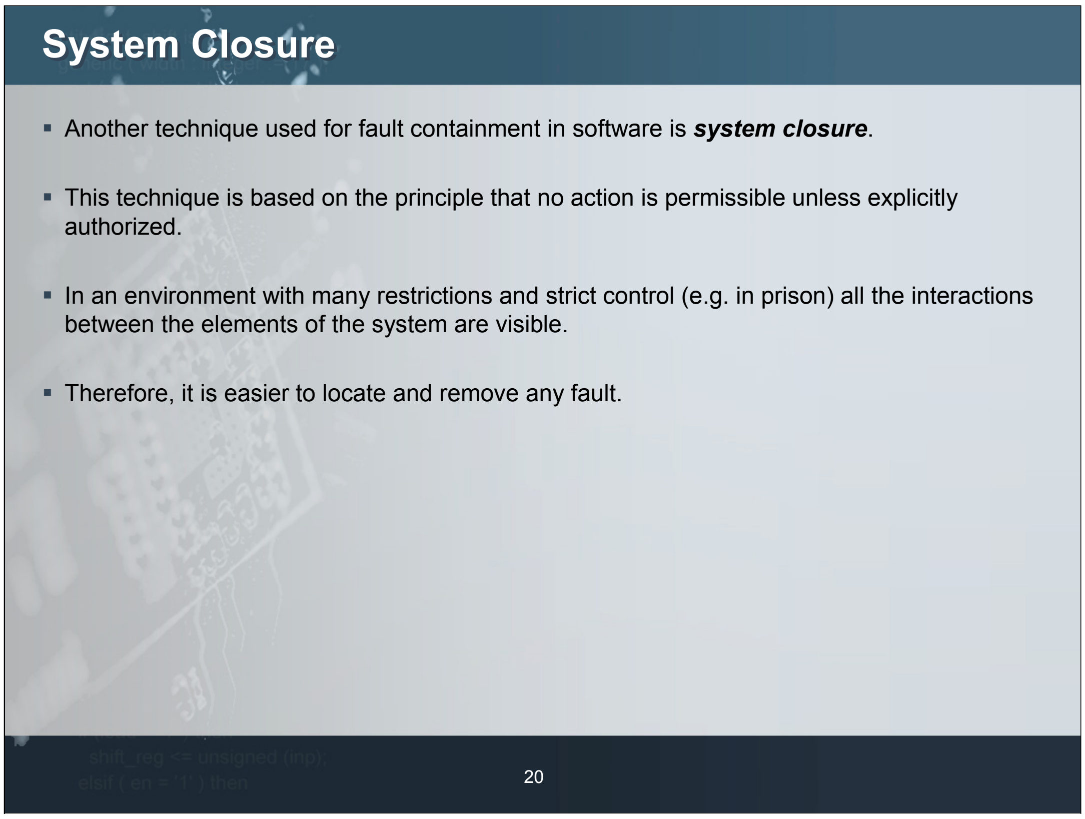
Modularization

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



Partitioning

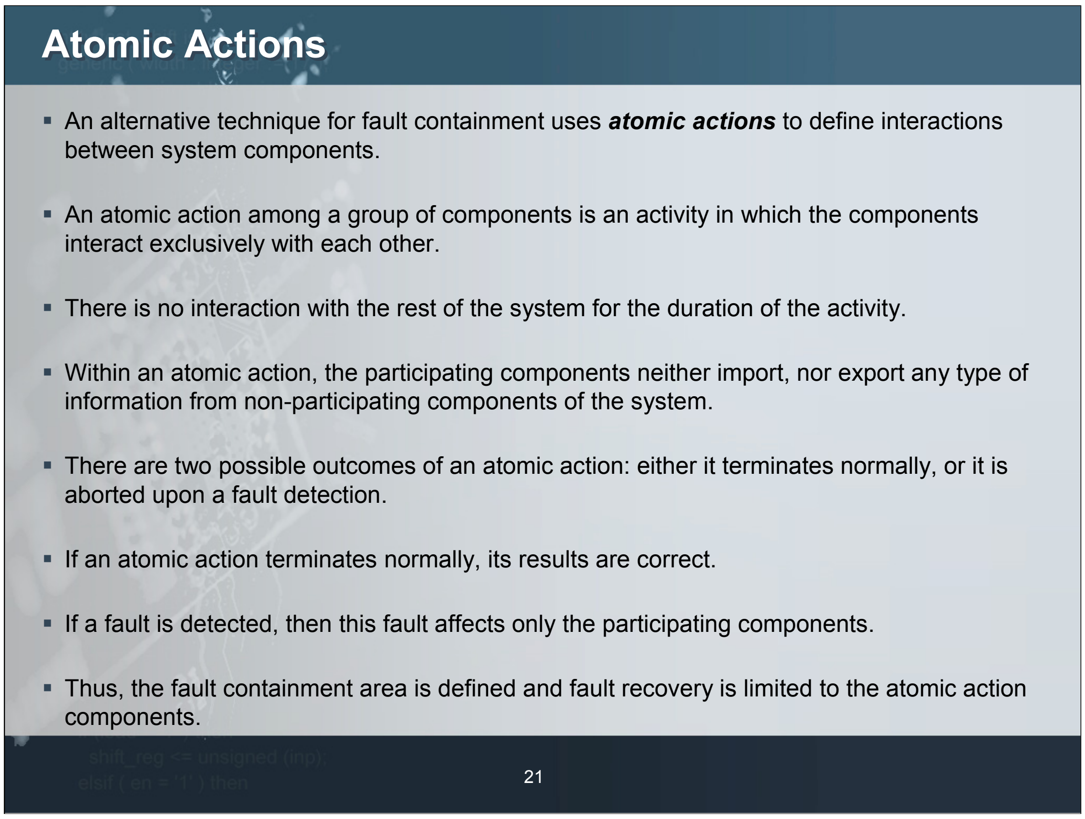
- The isolation between functionally independent modules can be done by ***partitioning*** the modular hierarchy of a software architecture in horizontal or vertical dimensions.
- Horizontal partitioning separates the major software functions into independent branches.
- The execution of the functions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



System Closure

- Another technique used for fault containment in software is ***system closure***.
- This technique is based on the 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.

20



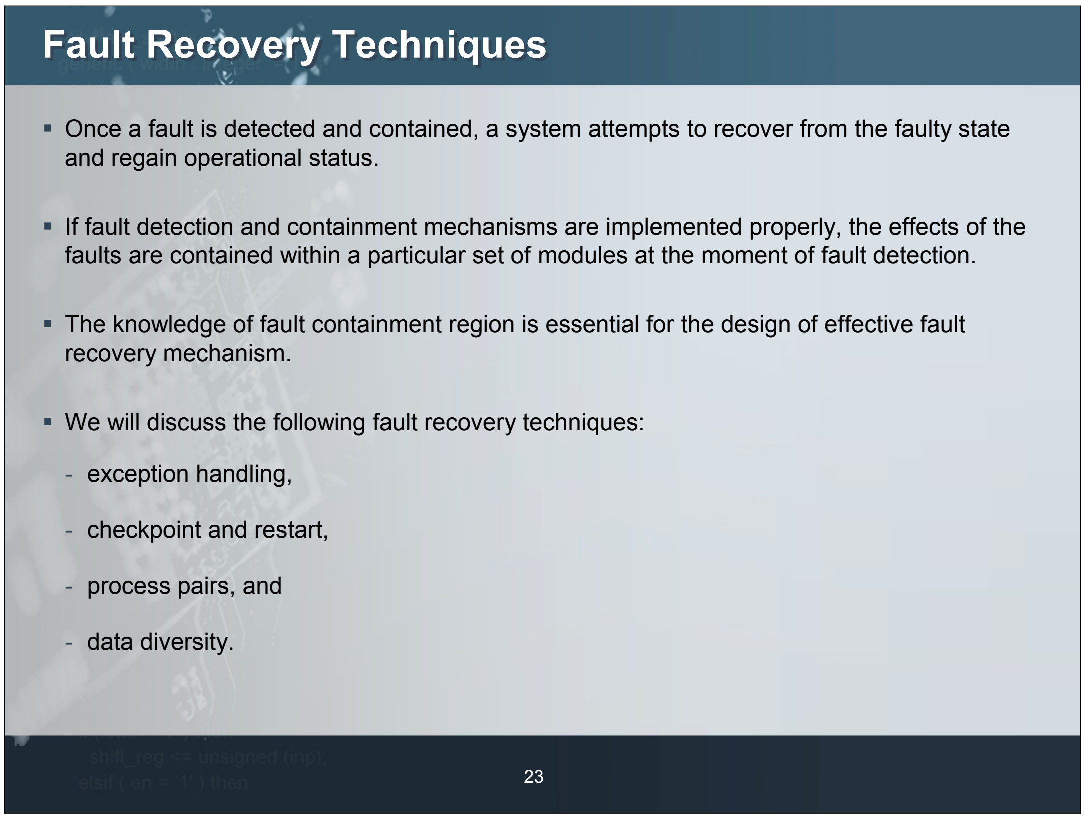
Atomic Actions

- 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.
- 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.
- Thus, the fault containment area is defined and fault recovery is limited to the atomic action components.

21

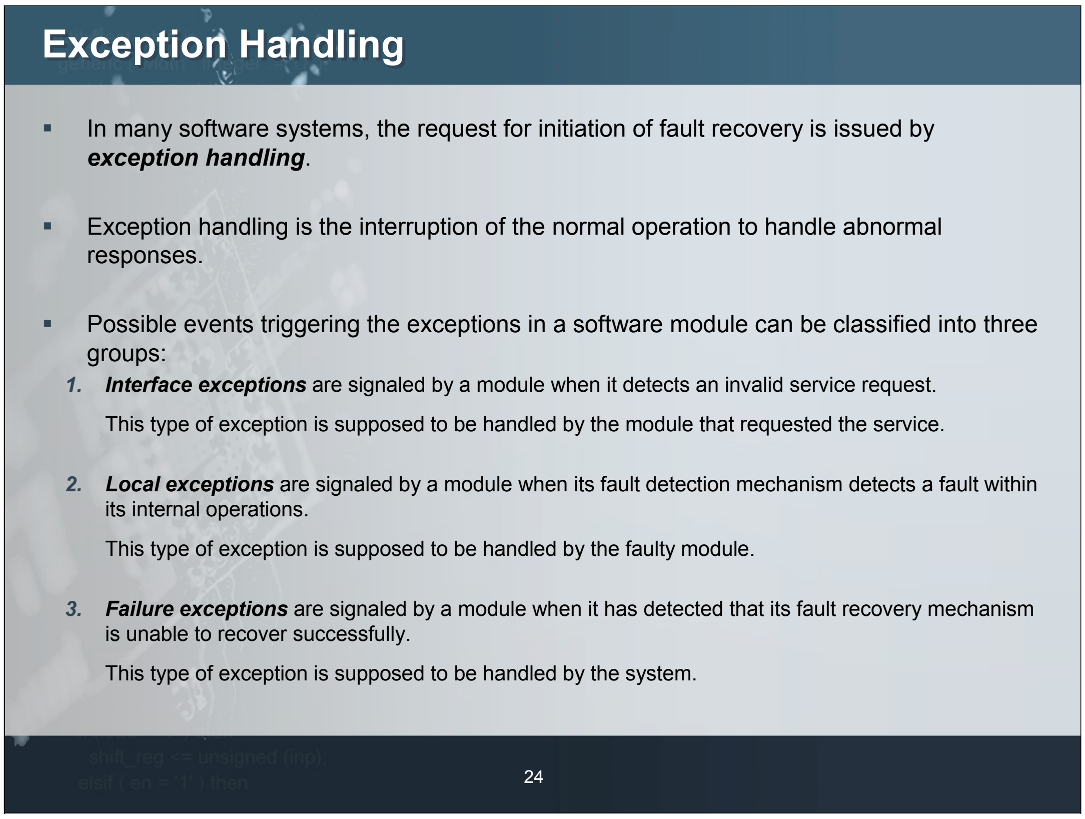
Software Fault Tolerance Techniques

Fault Recovery Techniques



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.
- We will discuss the following fault recovery techniques:
  - exception handling,
  - checkpoint and restart,
  - process pairs, and
  - data diversity.



Exception Handling

- In many software systems, the request for initiation of fault recovery is issued by ***exception handling***.
- Exception handling is the interruption of the normal operation to handle abnormal responses.
- Possible events triggering the exceptions in a software module can be classified into three groups:

1. ***Interface exceptions*** 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.

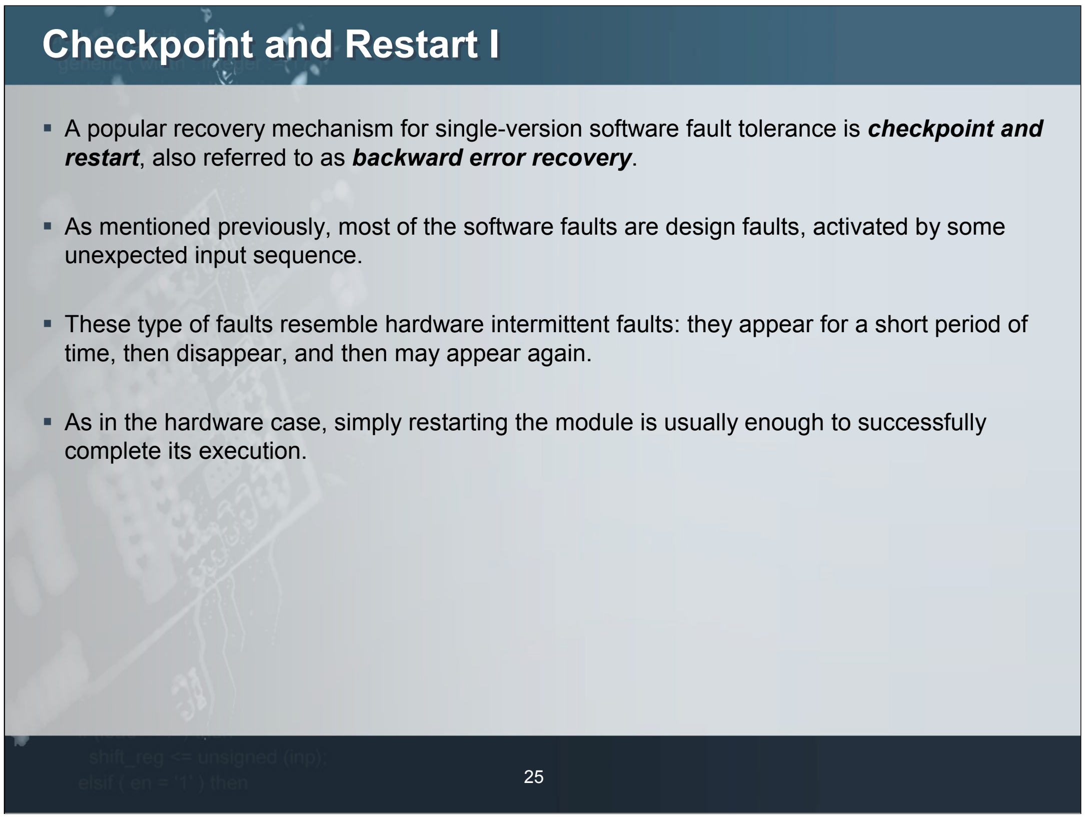
2. ***Local exceptions*** are signaled by a module when its fault detection mechanism detects a fault within its internal operations.

   This type of exception is supposed to be handled by the faulty module.

3. ***Failure exceptions*** are signaled by a module when it has detected that its fault recovery mechanism is unable to recover successfully.

   This type of exception is supposed to be handled by the system.

24



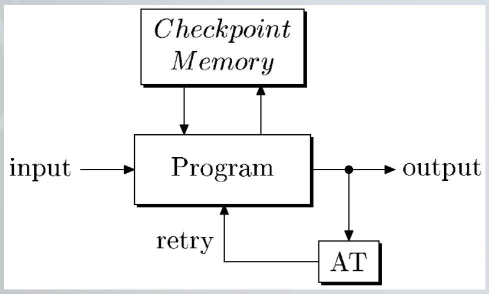
Checkpoint and Restart I

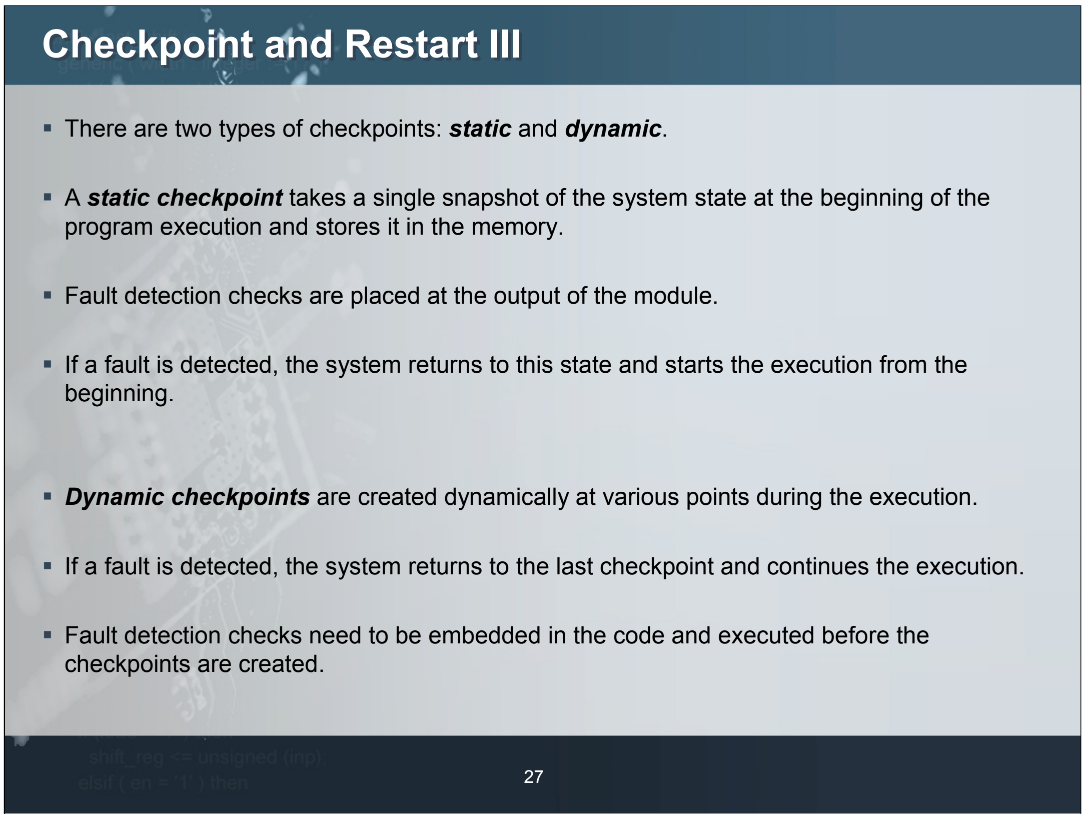
- A popular recovery mechanism for single-version software fault tolerance is ***checkpoint and restart***, also referred to as ***backward error recovery***.
- As mentioned previously, most of the software faults are design faults, activated by some unexpected input sequence.
- These type of faults resemble hardware intermittent faults: they appear for a short period of time, then disappear, and then may appear again.
- As in the hardware case, simply restarting the module is usually enough to successfully complete its execution.

25

Checkpoint and Restart II

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

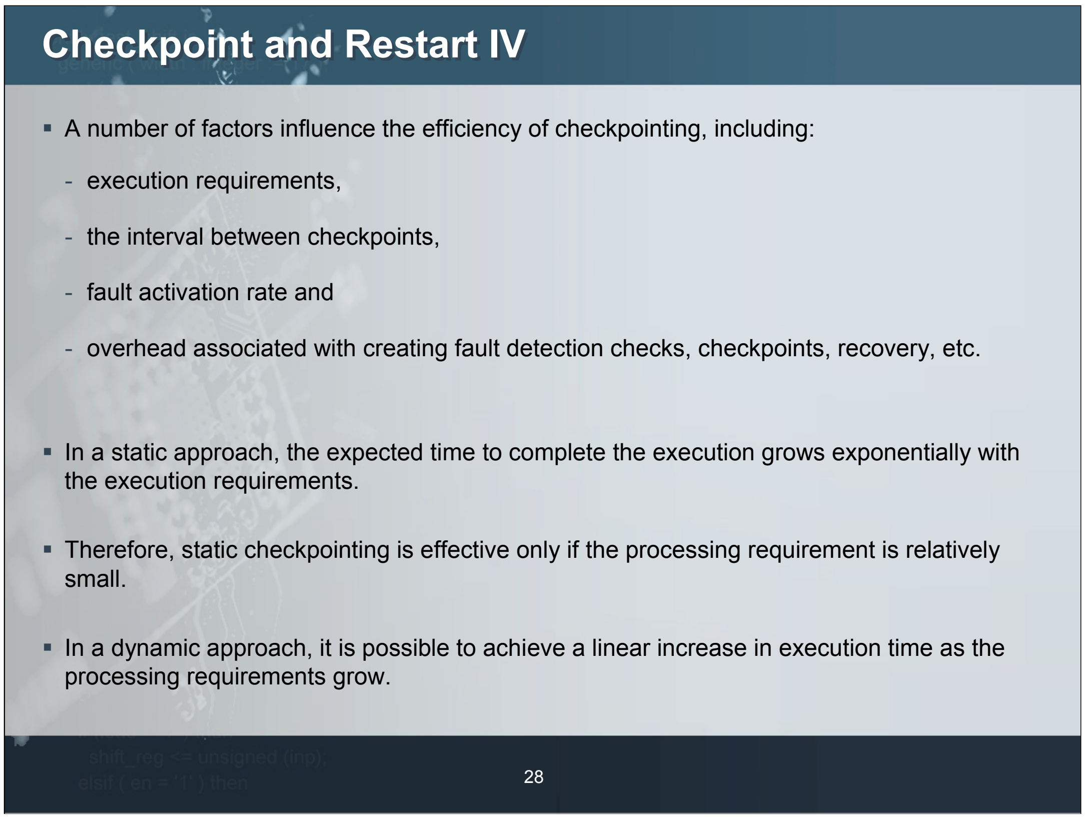




Checkpoint and Restart III

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

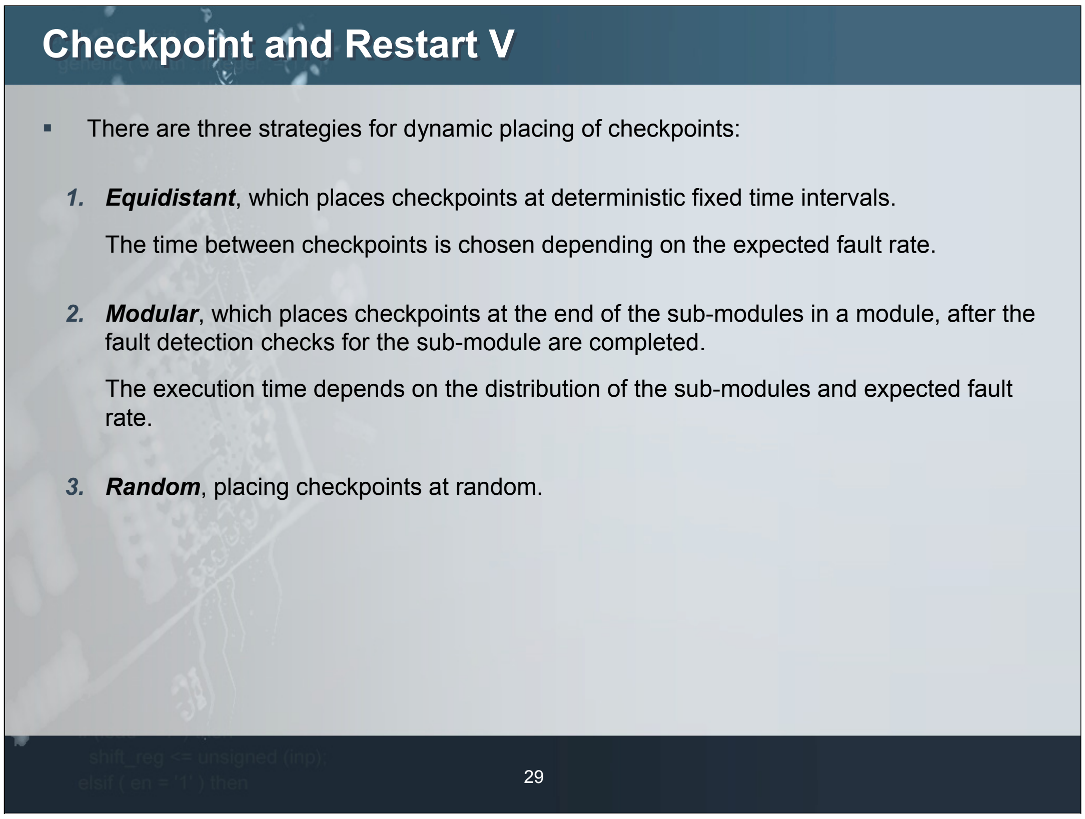
27



Checkpoint and Restart IV

- 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.
- In a 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 a dynamic approach, it is possible to achieve a linear increase in execution time as the processing requirements grow.

28



Checkpoint and Restart V

- There are three strategies for dynamic placing of checkpoints:

1. ***Equidistant***, which places checkpoints at deterministic fixed time intervals.

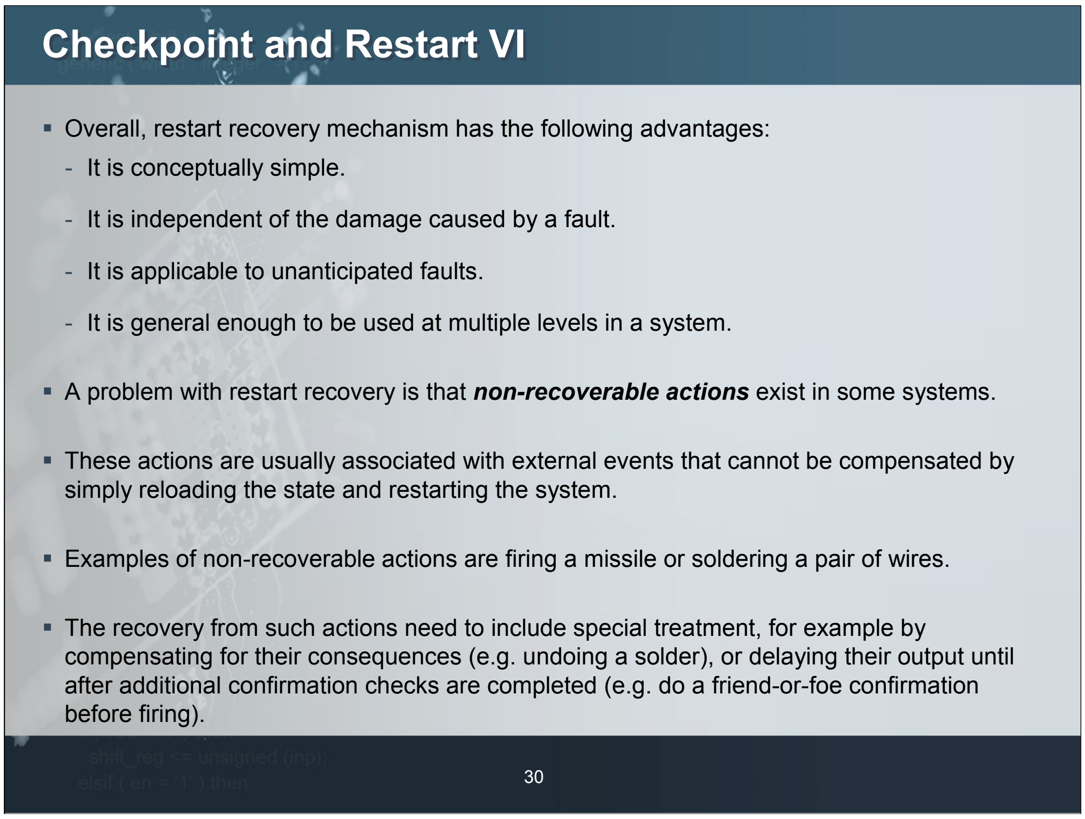
   The time between checkpoints is chosen depending on the expected fault rate.

2. ***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.

3. ***Random***, placing checkpoints at random.

29



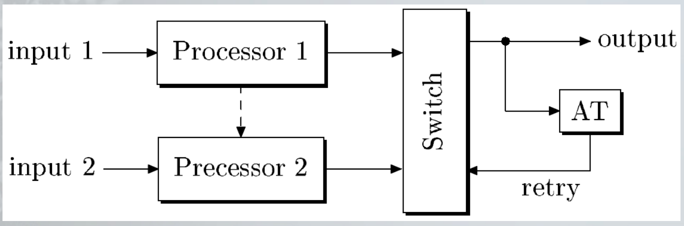
Checkpoint and Restart VI

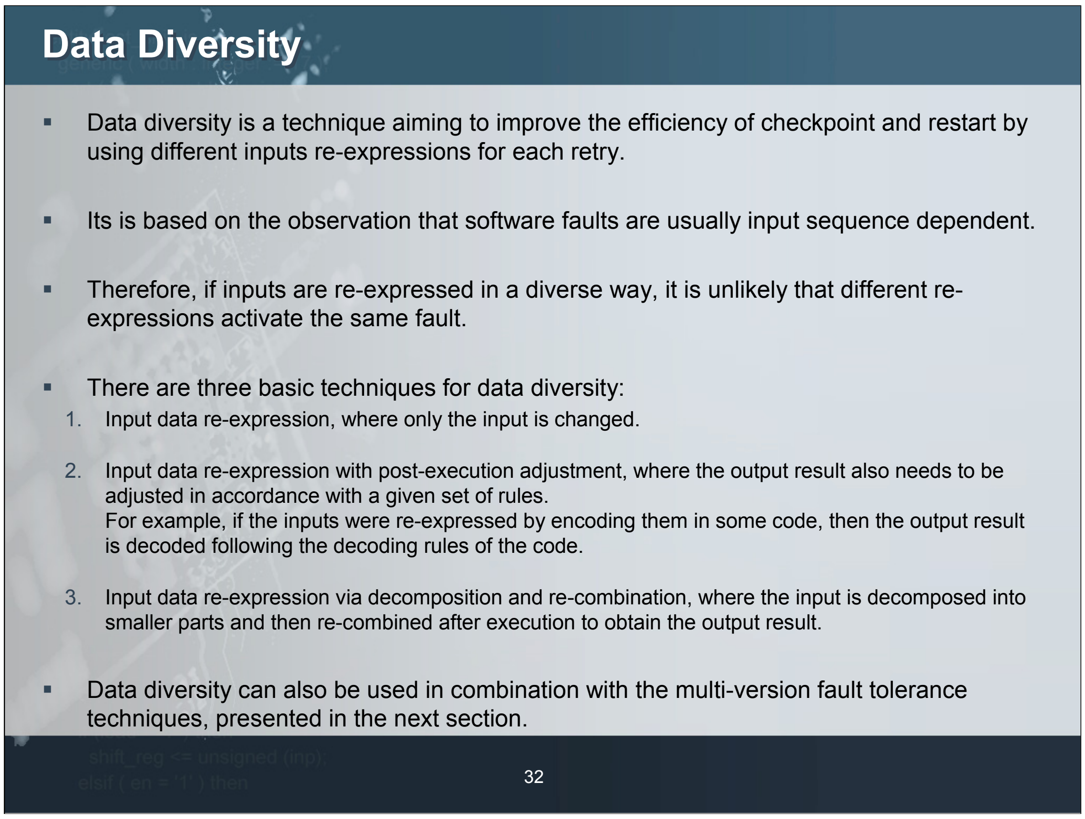
- Overall, restart recovery mechanism has the following advantages:
  - It is conceptually simple.
  - It is independent of the damage caused by a fault.
  - It is applicable to unanticipated faults.
  - It is general enough to be used at multiple levels in a system.
- A problem with restart recovery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

30

Process Pairs

- Process pair technique runs two identical versions of the software on separate processors (see Figure below).
- 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.
- The main advantage of process pair technique is that the delivery of service continues uninterrupted after the occurrence of the fault. It is therefore suitable for applications requiring high availability.





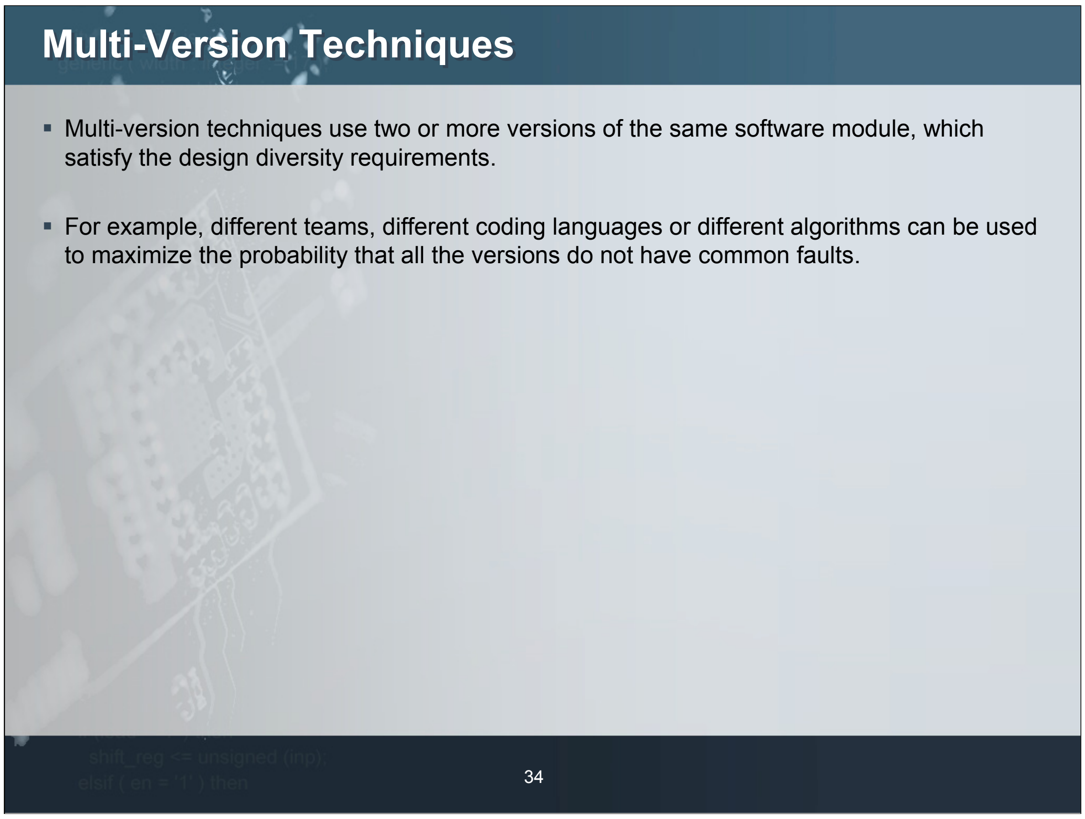
Data Diversity

- 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.
- There are three basic techniques for data diversity:
  1. Input data re-expression, where only the input is changed.
  2. 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.
  3. 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.
- Data diversity can also be used in combination with the multi-version fault tolerance techniques, presented in the next section.

32

Software Fault Tolerance Techniques

Multi-Version Techniques



Multi-Version Techniques

- Multi-version techniques use two or more versions of the same software module, which satisfy the design diversity 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.

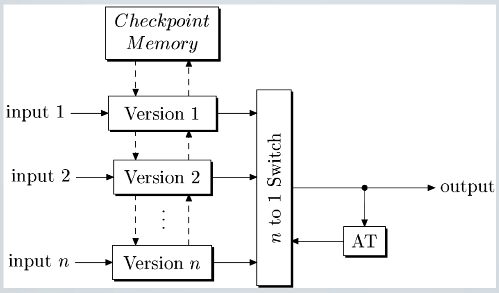
34

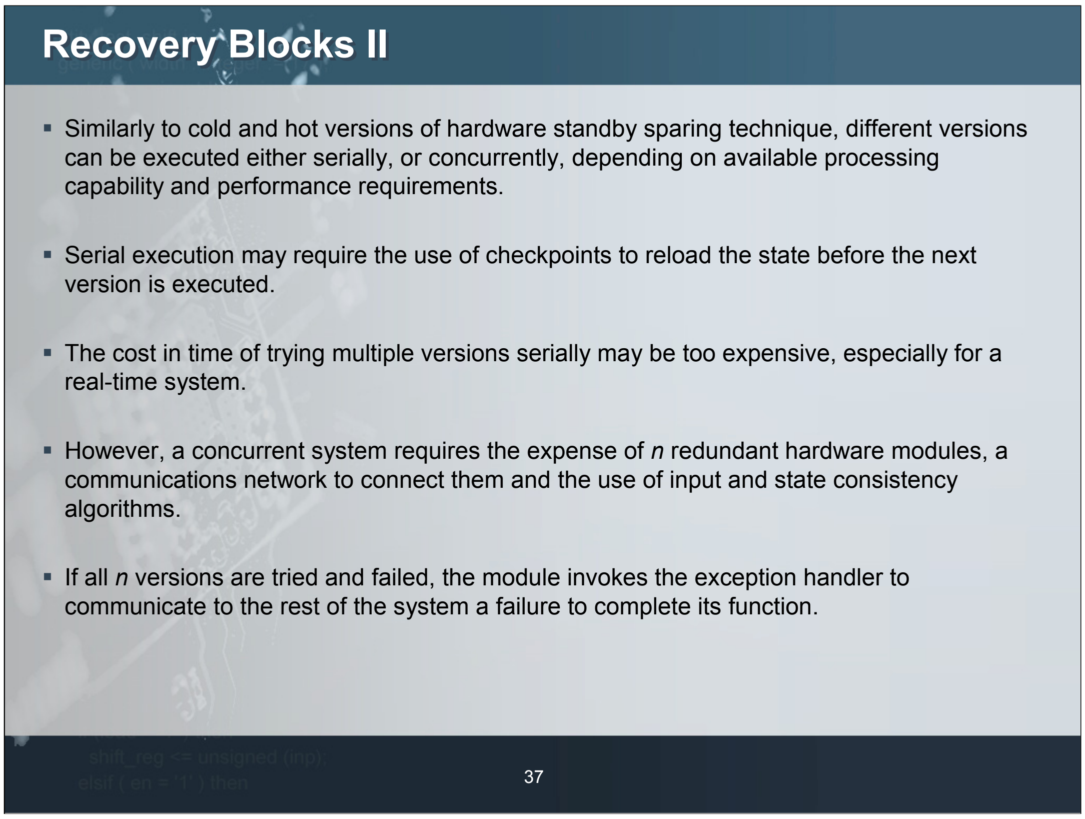
Software Fault Tolerance Techniques

Recovery Blocks

Recovery Blocks I

- The recovery blocks technique combines checkpoint and restart approach with standby sparing redundancy scheme.
- The basic configuration is shown in Figure on the right.
- Versions 1 to *n* represent different implementations of the same program.
- Only one of the versions provides the system's output.
- If an error is 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.
- 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.
- Checks can either be placed at the output of a module, or embedded in the code to increase the effectiveness of fault detection.

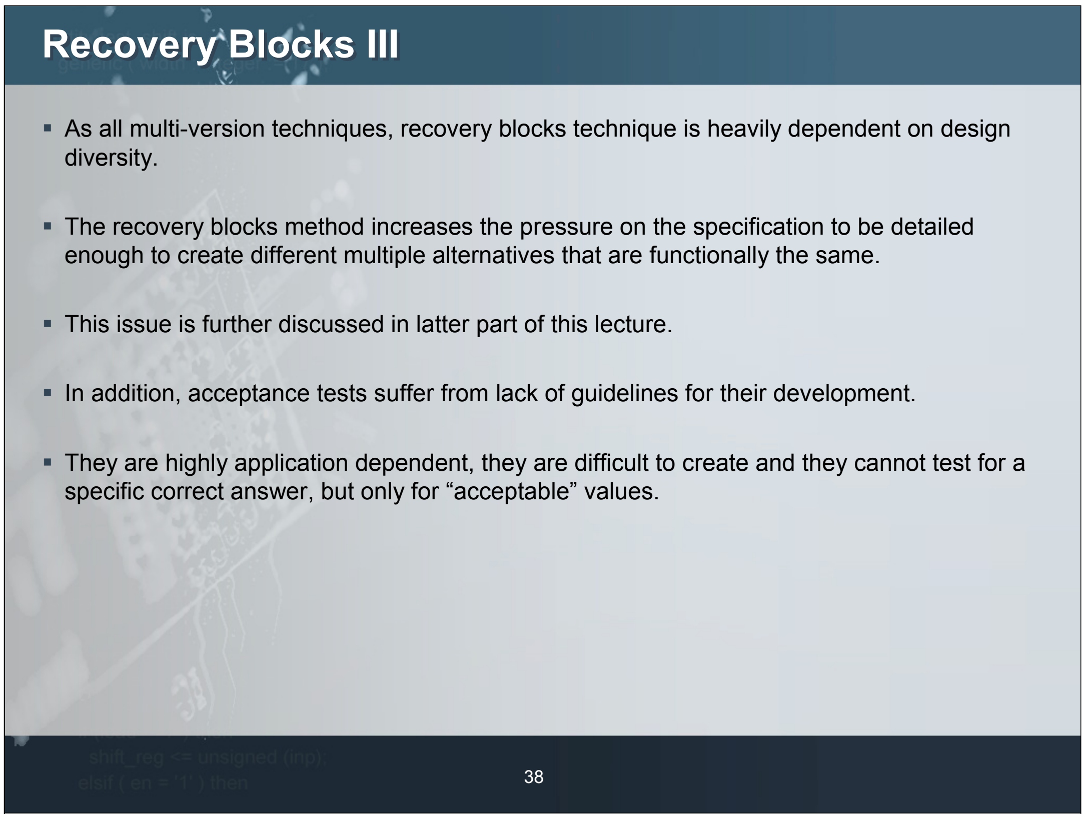




Recovery Blocks II

- 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.
- 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.

37



Recovery Blocks III

- 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 latter part of this lecture.
- In addition, acceptance tests suffer from lack of guidelines 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.

38

Software Fault Tolerance Techniques

N-Version Programming

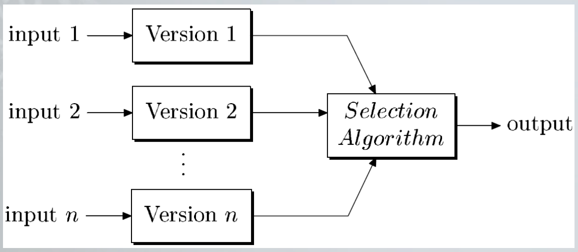
N-Version Programming I

- The *N*-version programming techniques resembles the *N*-modular hardware redundancy. The block diagram is shown in Figure below.

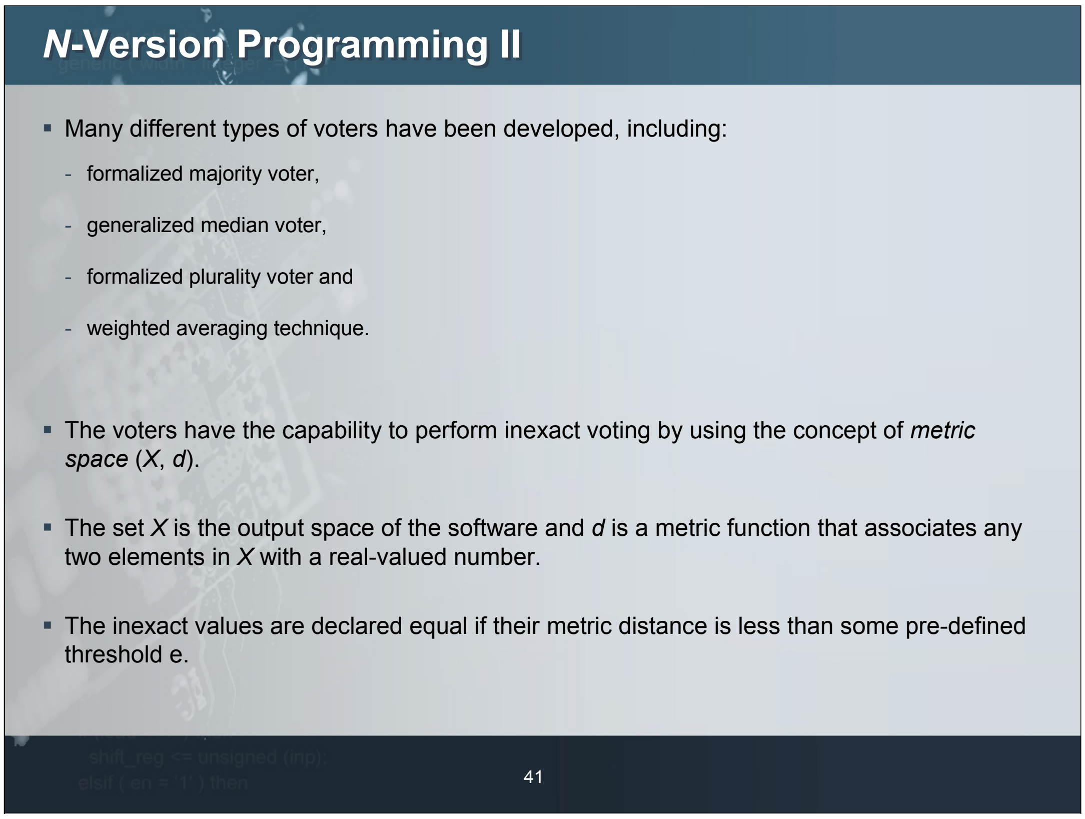
- It consists of *n* different software implementations of a module, executed concurrently.
- Each version accomplishes the same task, but in a different way.

- The selection algorithm decides which of the answers is correct and returns this answer as a result of the module's 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.



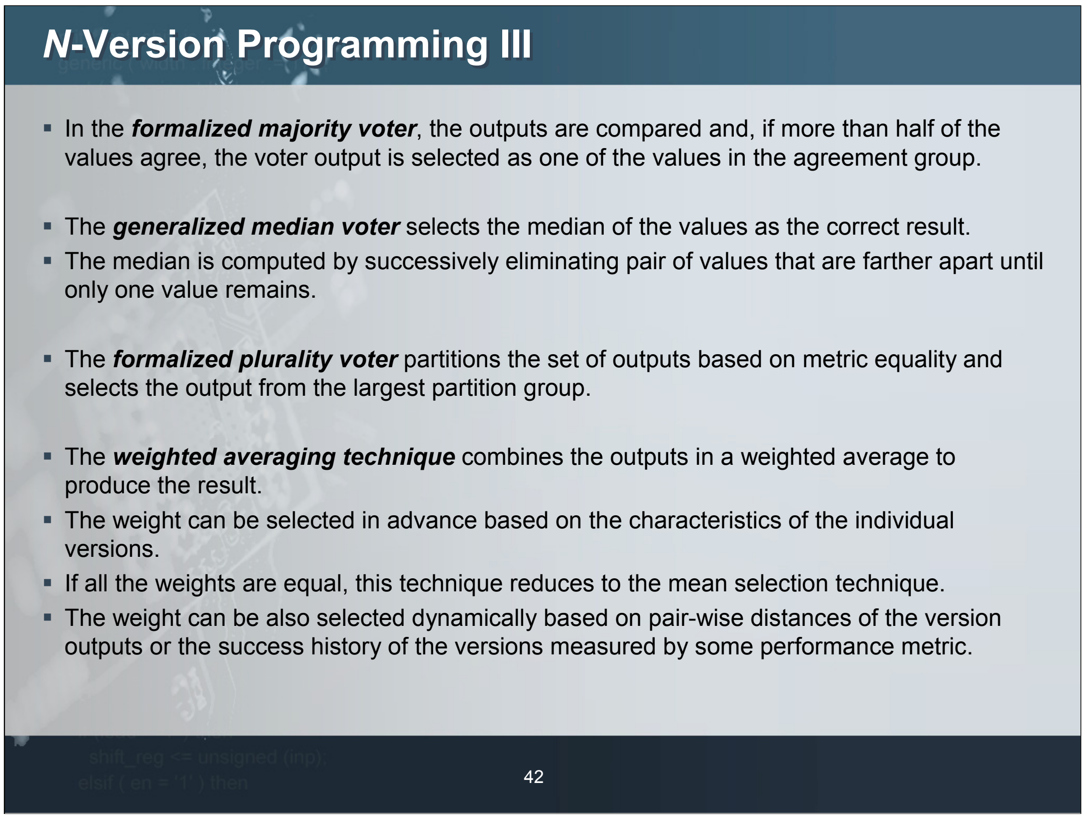
40



N-Version Programming II

- Many different types of voters have been developed, including:
  - formalized majority voter,
  - generalized median voter,
  - formalized plurality voter and
  - weighted averaging technique.
- 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.
- The inexact values are declared equal if their metric distance is less than some pre-defined threshold e.

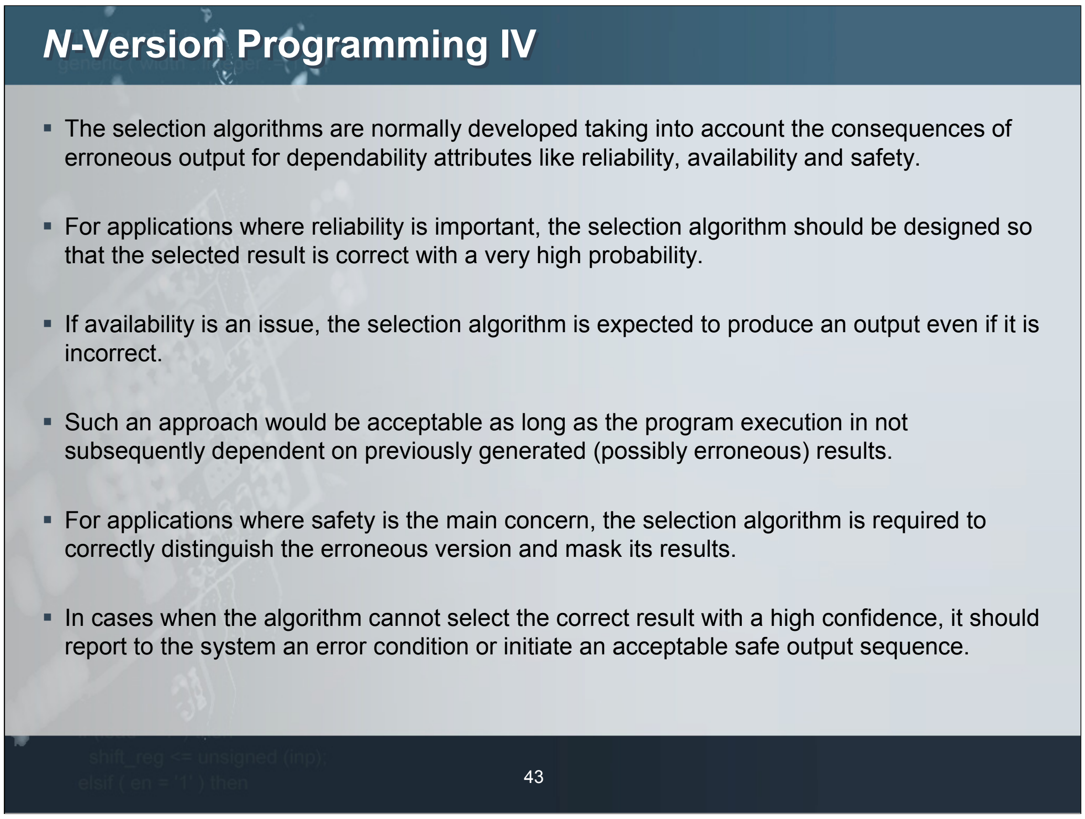
41



N-Version Programming III

- 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 apart until only one value remains.
- 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 mean selection technique.
- The weight can be also selected dynamically based on pair-wise distances of the version outputs or the success history of the versions measured by some performance metric.

42



N-Version Programming IV

- 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.

43

N-Version Programming V

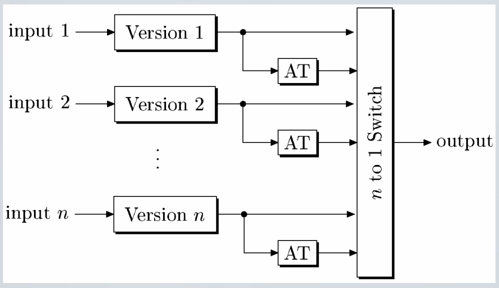
- *N*-version programming technique can tolerate the design faults present in the software if the design diversity concept is implemented properly.
- Each version of the module should be implemented in an as diverse as possible manner, including different tool sets, different programming languages, and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

Software Fault Tolerance Techniques

N Self-Checking Programming

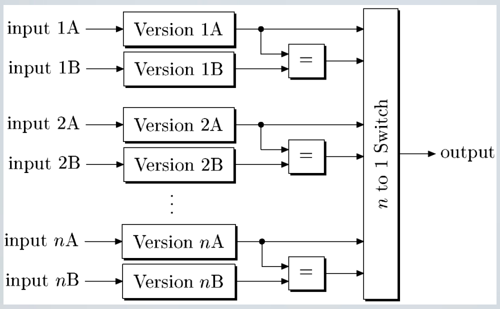
N Self-Checking Programming I

- *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.
- *N* self-checking programming using acceptance tests is shown in Figure on the right.
- 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.
- 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.



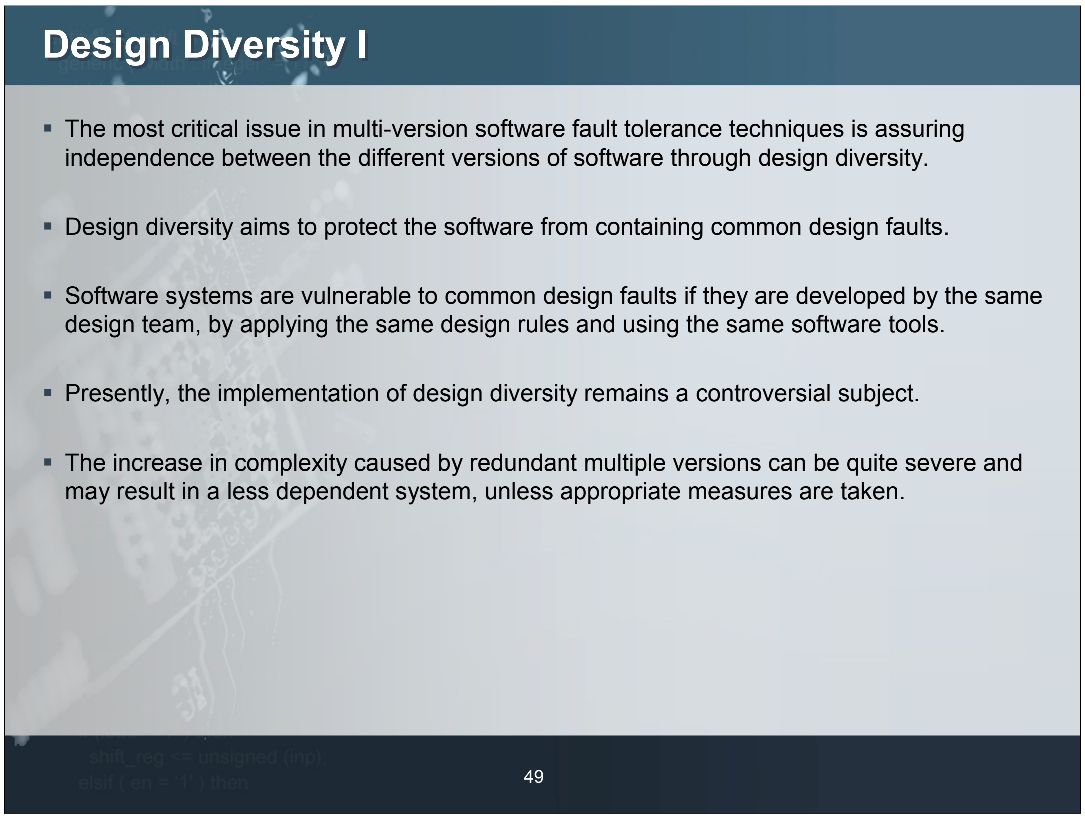
N Self-Checking Programming II

- *N* self-checking programming using comparison is shown in Figure on the right.
- 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.



Software Fault Tolerance Techniques

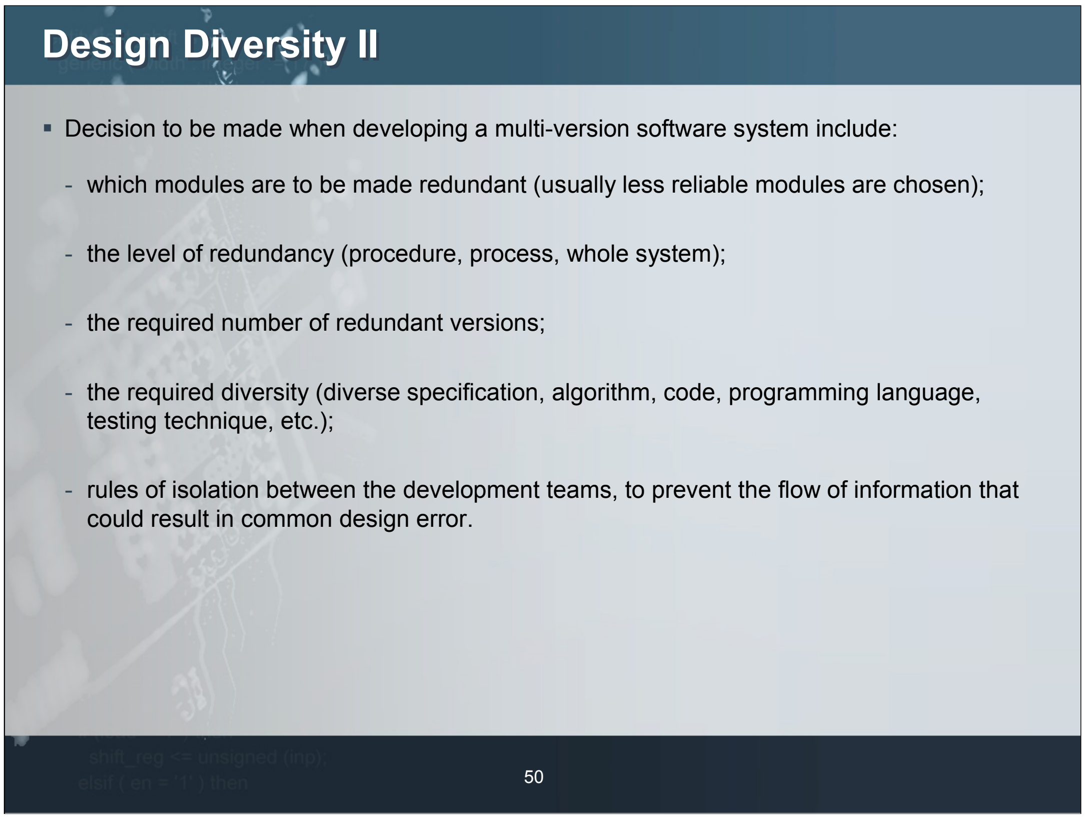
Design Diversity



Design Diversity I

- The most critical issue in multi-version software fault tolerance techniques is assuring independence between the different versions of software through design diversity.
- 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.
- Presently, the implementation of design diversity remains a controversial subject.
- The increase in complexity caused by redundant multiple versions can be quite severe and may result in a less dependent system, unless appropriate measures are taken.

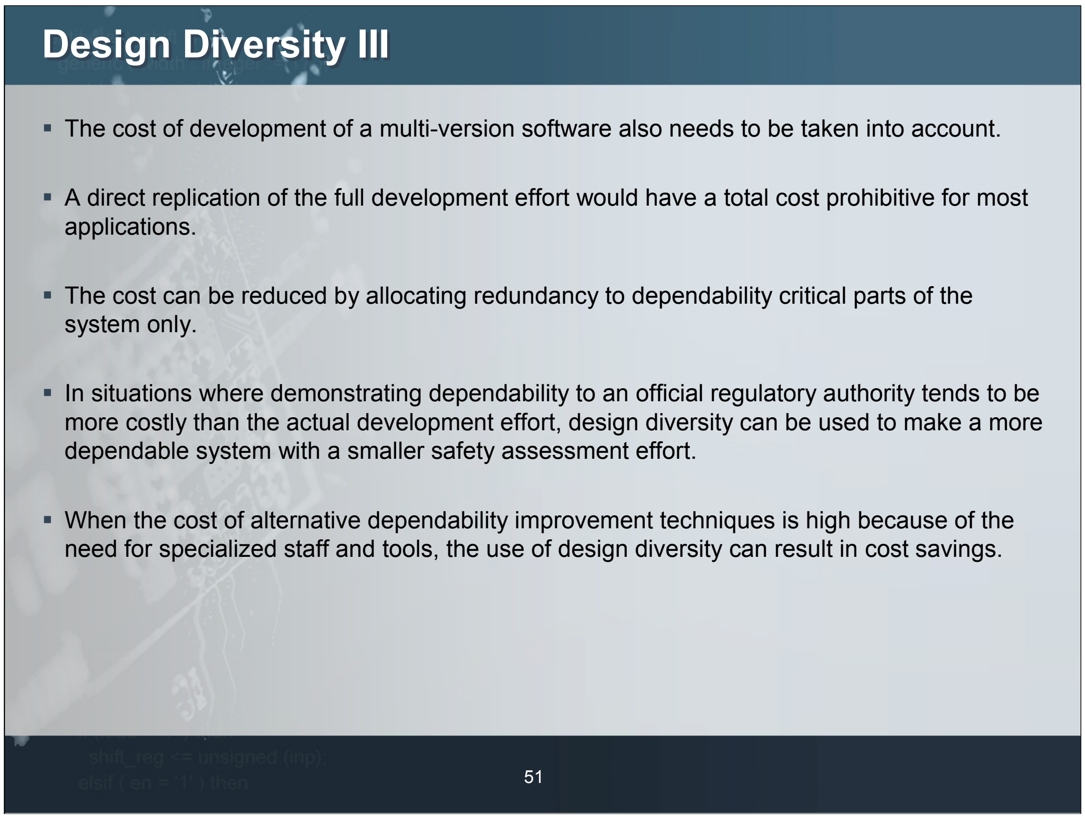
49



Design Diversity II

- 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.

50



Design Diversity III

- 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.
- In situations where demonstrating dependability to an official regulatory authority tends to be more costly than the actual development effort, design diversity can be used to make a more dependable system with a smaller safety assessment effort.
- When the cost of alternative dependability improvement techniques is high because of the need for specialized staff and tools, the use of design diversity can result in cost savings.

51