KUNGL TEKNISKA HÖGSKOLAN

VETENSKAP OCH KONST

International Master Program in System-on-Chip Design

Software redundancy

Software fault-tolerance

- Fault-tolerance in software domain is not as well understood as fault-tolerance in hardware domain
  - Controversial opinions exist on whether reliability can be used to evaluate software.
  - Software failures are mostly due to the activation of design faults by specific input sequences.
  - This makes the reliability of a software module dependent on the environment that generates input to the module over the time.
    - Ariane 5 rocket accident

Software fault-tolerance

- Many current techniques for software fault tolerance attempt to leverage the experience of hardware redundancy schemes
  – 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.

Problems

- Traditional hardware fault tolerance techniques were developed to fight
  – permanent components faults primarily
  – transient faults caused by environmental factors secondarily.
- They do not offer sufficient protection against design and specification faults, which are dominant in software.

Design diversity

- 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 has to be applied.
  - 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

Problems

- A software system usually has a very large number of states
  - a collision avoidance system required on most commercial aircrafts in the U.S. has 1040 states
- Software states do not exhibited adequate regularity to allow grouping them into equivalence classes.
  - Such regularity is common for digital hardware

Problems

- The large number of states implies that only a very small part of software system can be verified for correctness.
  - Traditional testing and debugging methods are not feasible for large systems.
  - Formal methods promise higher coverage, however, they are very complex
    - a specification using formal logic may be of the same size or even larger than the code.
- Due to incomplete verification, many design faults are not diagnosed and are not removed from the software

Single- and multi-version techniques

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

Redundancy allocation

- A number of possibilities has to be examined:
  - at which level the redundancy need to be provided
    - redundancy can be applied to a procedure, or to a process, or to the whole software system
  - which modules are to be made redundant
    - usually, the components which have high probability of faults are chosen to be made redundant.
- The increase in complexity caused by redundancy can be quite severe and may diminish the dependability improvement

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.
  - fault detection, fault containment and fault recovery
- 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.

Fault detection techniques

- The goal is to determine that a fault has occurred within a system.
- Various types of acceptance tests are used 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

Acceptance test

- Acceptance 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
  - structural checks

Timing checks

- Timing checks are applicable to system whose specification include timing constrains
- Based on these constrains, checks are 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.

Coding checks

- Coding checks are applicable to system whose data can be encoded using information redundancy techniques
- Usually 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

Reversal checks

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

Reasonableness checks

- Reasonableness checks use semantic properties of data to detect fault.
  - a range of data can be examined for overflow or underflow to indicate a deviation from system's requirements
    - maximum withdrawal sum in bank’s teller machine
    - address generated by a computer should lie inside the range of available memory

Structural checks

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

Fault containment techniques

- Fault containment is 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
- Techniques for fault containment:
  - modularization
  - partitioning
  - system closure
  - atomic actions

Modularization

- Software system is divided 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
  - by eliminating shared resources

Partitioning

- Modular hierarchy of a software architecture is partitioned 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

- System closure technique is based on a principle that no action is permissible unless explicitly authorized
- In an environment with many restrictions and strict control all the interactions between the elements of the system are visible
  - prison
- It is easier to locate and disable any fault.

Atomic action

- An atomic action among a group of components in an activity in which the components interact exclusively with each other.
  - no interaction with the rest of the system
- Two possible outcomes of an atomic action:
  - it terminates normally
  - it is aborted upon a fault detection
- Fault containment area is defined and fault recovery is limited to atomic action 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

Exception handling

- Exception handling is the interruption of normal operation to handle abnormal responses
- Possible events triggering the exceptions:
  - Interface exceptions
    - signaled by a module when it detects an invalid service request
  - Local exceptions
    - signaled by a module when its fault detection mechanism detects a fault
  - Failure exceptions
    - signaled by a module when it has detected that its fault recovery mechanism is enable to recover successfully

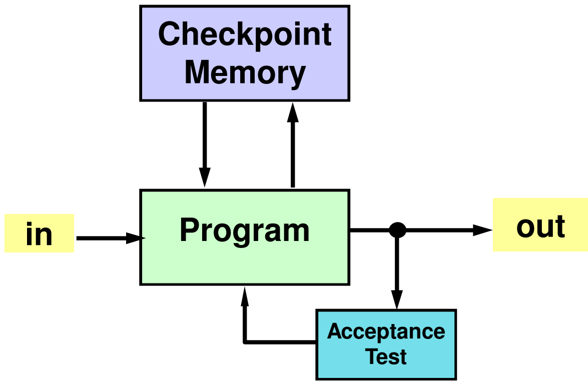
Checkpoint and restart

- Most of the software faults are design faults, activated by some non-tested or unexpected input sequence.
  - resemble hardware intermittent faults: appear for a short period of time, then disappear, and then may appear again.
- Simply restarting the module is usually enough to successfully complete its execution

Checkpoint and restart

- The module executing a program operates in combination with an acceptance test block which checks the correctness of the result
- If an 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 recovery



Static checkpoints

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

Dynamic checkpoints

- 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

Static vs. dynamic

- In static approach, the expected time to complete the execution grows exponentially with the execution requirements.
  - 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

Strategies for dynamic checkpointing

- Equidistant
  - places checkpoints at deterministic fixed time intervals
  - the time between checkpoints is chosen depending on the expected fault rate
- Modular
  - 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

Advantages of restart recovery

- Conceptually simple
- Independent of the damage caused by a fault
- Applicable to unanticipated faults
- General enough to be used at multiple levels in a system

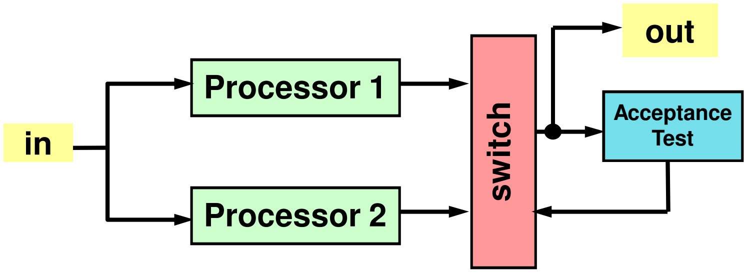
Problems of restart recovery

- Non-recoverable actions exist in some systems
  - these actions cannot be compensated by simply reloading the state and restarting the system
    - firing a missile
    - soldering a pair of wires
- The recovery from such actions can be done
  - by compensating for their consequences
    - undoing a solder
  - by delaying their output until after additional confirmation checks are completed
    - do a friend-or-foe confirmation before firing

Process pairs

- Two identical versions of the software are run on separate processors
- First the primary processor, 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

Process pairs



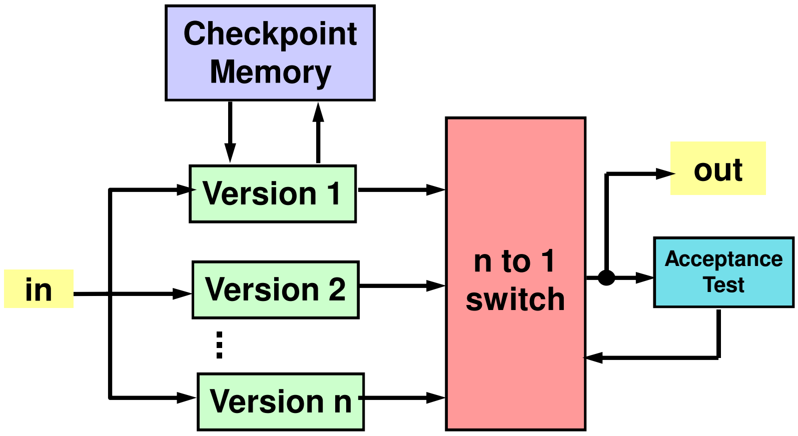
Multi-version techniques

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

- Combines checkpoint and restart approach with standby sparing redundancy scheme
- n different implementations of the same program
  - Only one of the versions is active
  - If an error if detected by the acceptance test, a retry signal is sent to the switch
  - The system in rolled back to the state stored in the checkpoint memory and the execution is switched to another module

Recovery blocks



Recovery blocks

- Similarly to cold and hot standby sparing, different version can be executed either serially, or concurrently
  - 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.
  - A concurrent system requires n redundant hardware modules, a communications network to connect them and the use of input and state consistency algorithms.

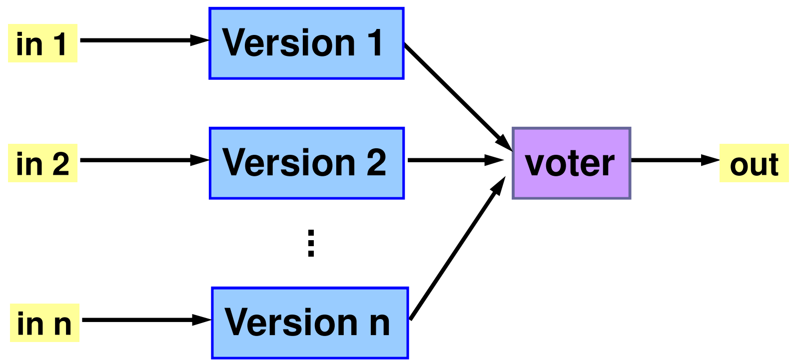
Recovery blocks

- 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 technique heavily depends on design diversity

N-version programming

- Resembles N-modular hardware redundancy
- N different software implementations of a module are executed concurrently.
- The selection algorithm (voter) decides which of the answers is correct
  - a voter is application independent
  - this is an advantage over recovery block fault detection mechanism, requiring application dependent acceptance tests

N-version programming



Voters

- There are many different types of voters:
  - formalized majority voter
    - selects majority
  - generalized median voter
    - selects the median of the values
  - formalized plurality voter
    - partitions the set of outputs based on metric equality and selects the output from the largest group
  - weighted averaging
    - combines the outputs in a weighted average

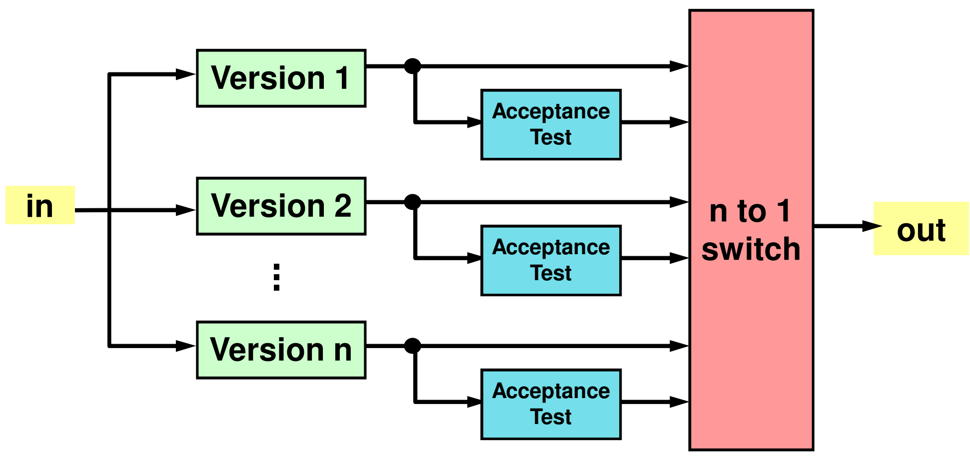
Voting

- The selection algorithms are normally developed taking into account the consequences of error
  - 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 it is incorrect
  - For applications where safety the main concern, the selection algorithm is required to correctly distinguish the erroneous version and mask its results

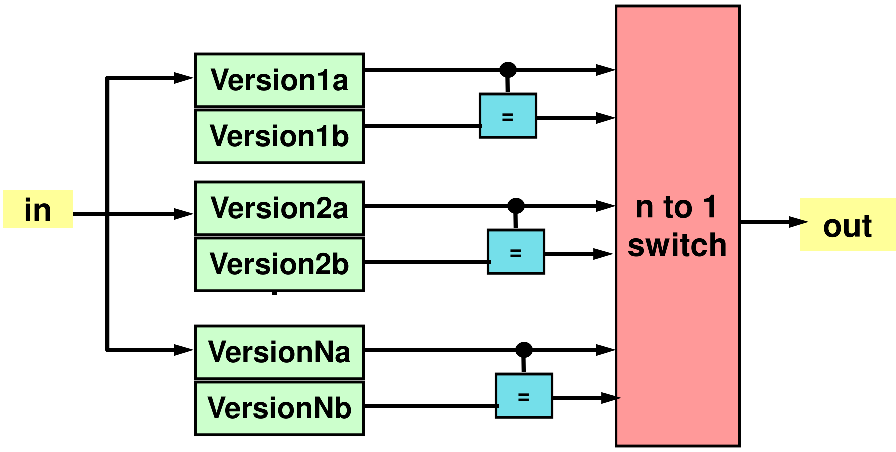
N self-checking programming

- N self-checking programming combines recovery block 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:
  - Lucent ESS-5 phone switch
  - Airbus A-340 airplane

N self-checking programming using acceptance tests



N self-checking programming using comparison



Comparison

- N self-checking programming using acceptance tests
  - The use of separate acceptance test for each version is the main difference of this technique from recovery blocks
- N self-checking programming using comparison
  - resembles triplex-duplex hardware redundancy
  - An advantage over N self-checking programming using acceptance tests is that the application independent decision algorithm 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
- 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

- 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
  - rules of isolation between the development teams

Software testing

- Software testing is the process of executing a program with the intent of finding errors
- Two types of software testing:
  - Functional testing compares test program behavior against its specification
  - Structural testing checks the internal structure of a program for errors

Structural testing

- The effectiveness of structural testing is expressed in terms of test coverage metrics which measure the fraction of code exercised by tests
  - Statement coverage
  - Branch coverage
  - Path coverage

Statement coverage

- Statement coverage requires that each executable statement of a program is followed during a test
- Advantages:
  - Can be applied directly to object code and does not require processing source code
- Disadvantages:
  - Insensitive to some control structures, logical AND or OR operators, and switch labels

Example

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

- 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

Branch Coverage

- Branch coverage requires that each branch of a program is executed at least once during a test
- Advantages:
  - relative simplicity
- Disadvantages:
  - might miss some errors

Example

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

- 100% branch coverage can be achieved by two tests:
  - both condition1 and condition2 evaluate true
  - both condition1 and condition2 evaluate false
- However, the error which occur when condition1 evaluates true and condition2 evaluates false is not detected

Path coverage

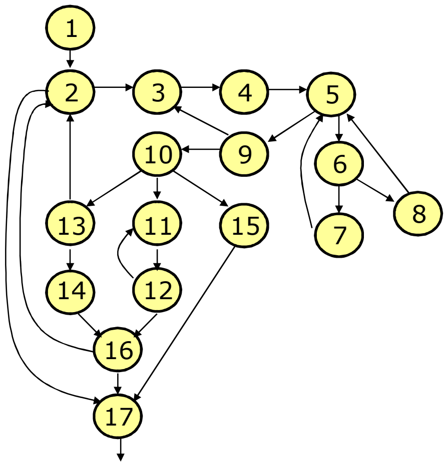
- Path coverage requires that each of the possible paths through the program is followed during a test
- The most reliable metric, however, not applicable to large programs
  - the number of paths is exponential to the number of branches
- 100% branch coverage is a requirement of most software standards

Preliminaries

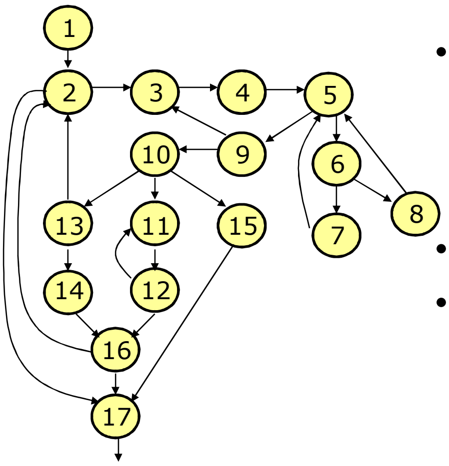
- A flowgraph is a directed graph $G = (V,E,entry,exit)$ where
  - V is the set of vertices representing basic blocks of the program
  - $E \subseteq V \times V$ is the set of edges connecting the vertices
- entry and exit are two distinguished vertices of V

Example

```
b1;
while(b2)
  for(b3)
    b4;
    for(b5)
       if(b6) b7;
       else   b8;
    if(b9) break;
  switch(b10) {
    case 1: while(b11) b12;
    case 2: if(b13) b14;
            else continue;
    default: b15;
             break;
  b16;
b17;
```

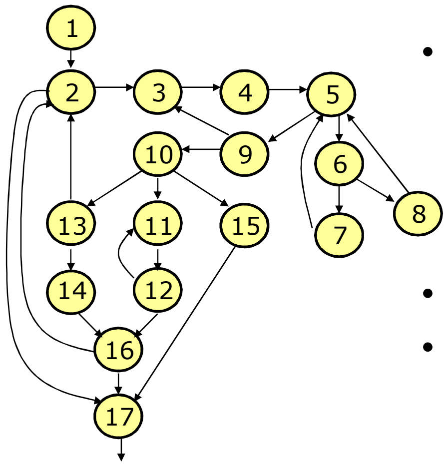


Pre-dominators



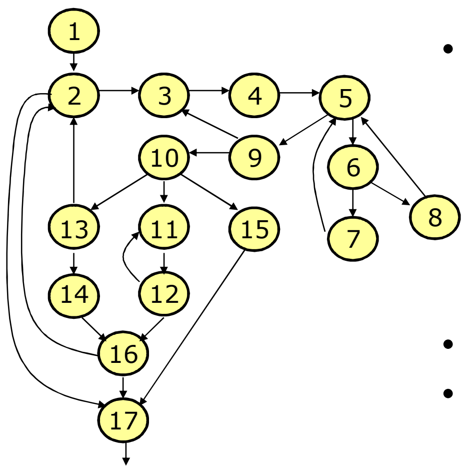
- A vertex v pre-dominates a vertex u if every path from entry to u contains v
- 4 pre-dominates 5
- 6 pre-dominates 7 and 8

Post-dominators



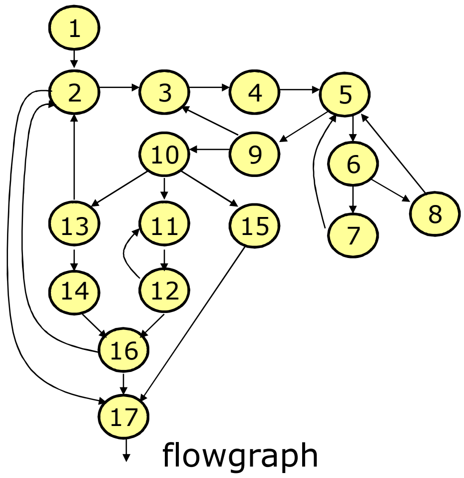
- A vertex v post-dominates a vertex u if every path from u to exit contains v
- 9 post-dominate 5
- 5 post-dominate 6

Immediate dominators

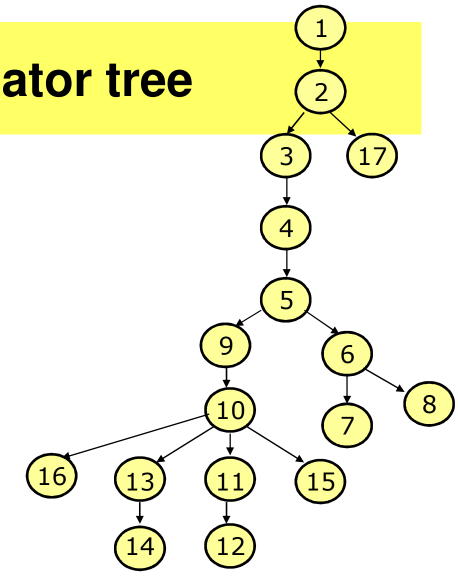


- Vertex v is the immediate pre-dominator of u, if v pre-dominates u and every other pre-dominator of u pre-dominates v
  - 1,2,3,4 pre-dominate 5
  - 4 is immediate
- unique
- edges (idom(v),v) form a tree rooted at entry

Pre-dominator tree

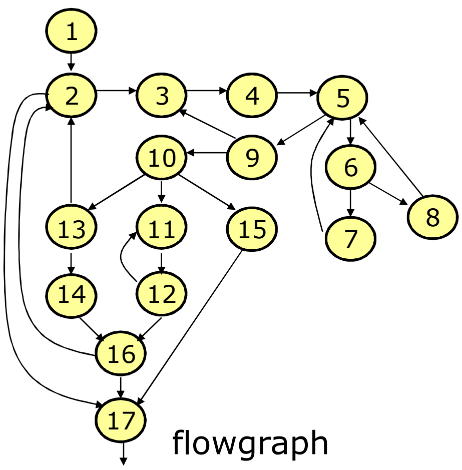


flowgraph

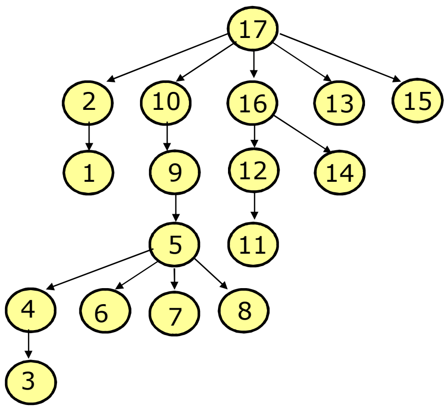


pre-dominator tree

Post-dominator tree



flowgraph



post-dominator tree

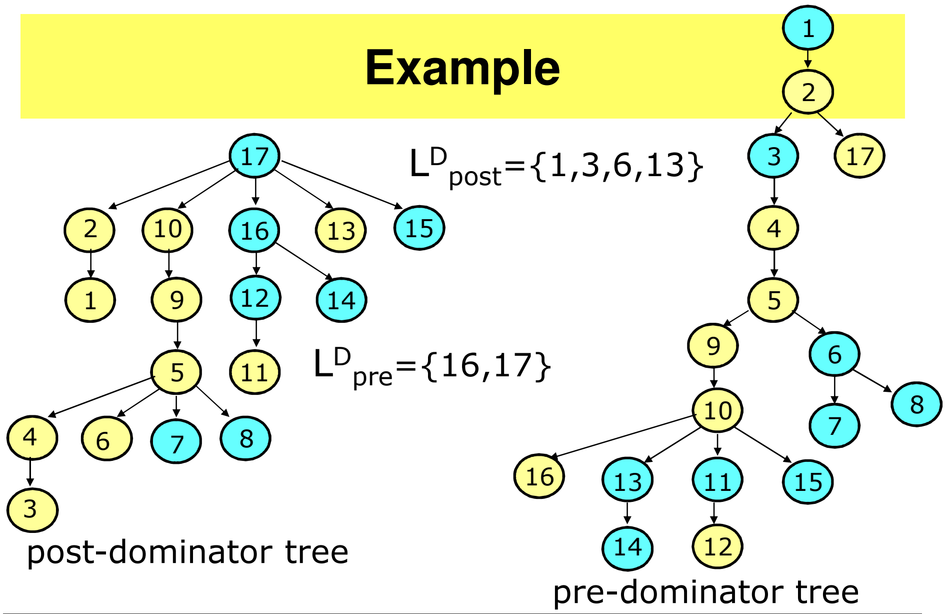
Statement Coverage

- We present a technique for finding a subset of flowgraph vertices, called kernel
- any set of tests which executes all vertices of the kernel executes all vertices of the flowgraph
- 100% statement coverage can be achieved by constructing a set of tests for the kernel

Notation

- L_{pre} denotes the set of leaf vertices of the pre-dominator tree of G
- L^D_{pre} contains all vertices of L_{pre} which post-dominate some vertex of L_{pre}
- L_{post} denotes the set of leaf vertices of the post-dominator tree of G
- L^D_{post} contains all vertices of L_{post} which pre-dominate some vertex of L_{post}

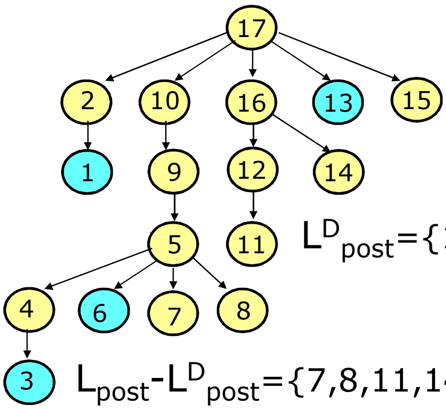
Example



Properties of kernels

- The sets L_{pre} - L^D_{pre} and L_{post} - L^D_{post} are minimum kernels for G
- Minimum kernels can be computed in O(|V|+|E|) time

Example

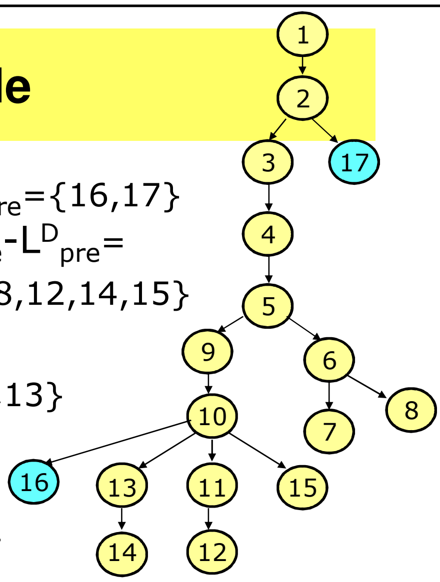


$L^D_{pre}=\{16,17\}$

$L_{pre}-L^D_{pre}=\{7,8,12,14,15\}$

$L^D_{post}=\{1,3,6,13\}$

$L_{post}-L^D_{post}=\{7,8,11,14,15\}$



post-dominator tree

pre-dominator tree

Branch Coverage

- The kernel-based technique can be similarly applied to branch coverage by constructing pre- and post-dominator trees for the edges of the flowgraph instead of for its vertices
- 100% branch coverage can be achieved by constructing a set of tests for the kernel

Summary of structural testing

- Technique for structural testing based on kernel computation
- Any set of tests which executes all vertices of the kernel executes all vertices of the flowgraph
- 100% coverage can be achieved by constructing a set of tests for the kernel

Summary

- Basic techniques for achieving fault tolerance
  - hardware redundancy
  - information redundancy
  - time redundancy
  - software redundancy
- Often a combination of techniques is used, depending on application

Summary

- It is important to be able to compare how good are two or more different approaches for a particular application, without implementing them
- Results of comparison lead to trade-offs and modification of the design
- This is done using evaluation methods

Next lecture

- Fault tolerance in VLSI systems (not covered in the text book)