so far have looked at reliability as hardware reliability

however, reliability of the overall system is actually a product of the hardware, software, and human reliability

in this section, consider software reliability

overall concept of software engineering includes engineering and management technologies used to develop cost-effective, schedule-meeting software

one of those technologies is software reliability measurement and estimation as used to measure and predict the probability that the software will perform its intended function as specified without error during the given period of time

such techniques are part of the design process for reliable software

in contrast, software reliability is set of fail-safe design techniques for assuring that if the program crashes computer will recover to reinitialize and restart the program

important to note that if same input given to a program repeatedly, then the result will be the same each time assuming no hardware, power or OS errors

software, unlike hardware, does not degrade over time

software faults result of design faults or ‘bugs’ which are caused by humans

unlike hardware, replicating the software won’t help

for software with large range of values for inputs, impossible to test all input combinations

software development life cycle is the model followed to develop reliable software typically using OOP & design diagrams called universal modeling language (UML)

steps shown in Table 5.1
- note that if using TMR for hardware FT and there is an error in the software, all three modules will give the same erroneous output

**Software Redundancy for Hardware Checks**

- before looking at main SWFT techniques, note that can use software to complement hardware FT

- for example, can add some lines of code to check magnitude of given signal or use small routine to periodically test memory by writing to and reading from, specific memory locations

- when know acceptable range of values for particular parameter, such as bank withdrawal, acceptable memory address, can check to be sure value is within range

- can periodically test ALU by having processor execute specific instructions on given data and compare to known results

- when have multiple processors, can require each to write certain values and compare

**Rollback and Recovery**

- rollback and recovery is actually a software technique for protection against hardware errors

- cost-effective technique against transient and intermittent faults

- checkpoint data is state at most recent checkpoint and is stored in memory

- active data is that actually in processor registers and cache

- main memory assumed stable through use of ECCs, duplexing and battery back-up

- rollback accomplished by discarding all modified registers and cache lines and restoring them to previous state used for checkpointing

- very complicated process to implement correctly and accurately
Motivation

- goal of SWFT is to guarantee that software system does not fail even with faulty software modules

- since replicating software (unlike hardware) will not help, need to have *design diversity* where modules have different designs in order to deal with design faults

- two main techniques:

  o N-version programming
  o Recovery Block approach

- require some kind of framework to embed FT technique

- results from software faults equally significant with hardware faults when considering such safety-critical applications as passenger aircrafts, nuclear power plants, launch controls for strategic missiles, etc.

- assume that in any practical program there are faults and hence must have means to deal with them if resulting errors have serious consequences

- *robustness* is defined by IEEE as the extent to which software can continue to operate correctly despite invalid inputs, e.g. out-of-range inputs, different format of input than is acceptable

- can use a version of *temporal redundancy* for software as well as hardware in that the fault may be caused by a busy or noisy communication channel, full buffers or power supply transient fault, and, all of these would execute correctly if executed a second time

- *software diversity* is the use of multiple versions of functional process, whether static (N-version programming) or dynamic (recovery-block)

N-version Programming

- independent generation of $N \geq 2$ functionally equivalent programs, called versions, from identical specifications

- for $N = 2$, detection but doesn’t really allow for continued operation
if disagreement, three options:
  o retry or re-start
  o transition to predefined ‘safe state’ with possible later retry
  o reliance on one version determined more reliable

if critical software, then above not acceptable

with N > 2, use majority voting

typically have N = 3 such that:
  o 3 independent programs with identical output formats
  o acceptance program that evaluates output and chooses output
  o control to implement above and give output to other modules

major design issue is how often comparison and voting should be done

if large space between comparisons, then performance penalty reasonable but can have long wait between synchronizations

if small space between comparisons, less likely to have independence among versions and higher performance penalty

thus, reasonable to be in between two extremes

note that voting, selection of output, and, providing output to another module must also all be FT

recommended that following approach used to obtain independent versions:
  o each programmer works from same requirements
  o each programmer or group works independently of others, with messages sent through contracting agency as only means of communication between groups
  o each version of software subjected to same acceptance tests
still can have dependence among errors due to:
- identical misinterpretation of requirements
- identical, incorrect treatment of boundary conditions
- identical incorrect design of difficult sections

most testing of 3-version programming has been in the research community but the following have either used it or proposed its use:
- space shuttle flight control software
- slat-and-flap control system of A310 Airbus
- point switching, signal control, and traffic control in part of the Swedish State Railway
- proposed for various nuclear reactor systems

repeatedly had the problem of correlated failures in 2 or more versions due to common statement of requirements, which could be a problem for all such fault tolerance and fault avoidance techniques

another problem is in numerical outputs where results differ slightly due to the finite precision arithmetic used

in dealing with problems such as finding roots of nth order equations, can be multiple correct solutions, depending on the particular technique used

some versions may take significantly longer to run, but must wait for all to complete, so performance is dominated by worst performing module

also can be interdependence among programmers on different teams as in general programmers have similar educational backgrounds using same programming language and common specifications

perfect voting assumes none of these problems exist

overall, cost of multiple versions can be very significant, especially as cost of programmers increases
Space Shuttle Example

- space shuttle is an example of both hardware and software reliability applied to the flight control system
- when in orbit, flight control system must maintain vehicle’s altitude in 3 dimensional inertial space
- space shuttle uses combination of large and small gas jets oriented about 3 axes to give needed rotations
- even more crucial during re-entry
- considerable hardware redundancy in sensors, gas jets, etc.
- Figure 5.19 gives overview of flight control system
- 5 identical computers and 2 different software programs
- computers A-D and software A are the Primary Avionics Software System (PASS), a 2-out-of-4 system
- if fault detected in one of the 4 systems and confirmed by continued analysis and with disagreement with other 3 systems and by data sent to Ground Control, astronauts remove this system and overall is now TMR
- consequently, two failures can be handled definitely, and probably another fault due to all the testing
- Backup Flight Control System (BFS) is on 5th computer with independent software
- two versions of software developed independently by two different groups, IBM Federal Systems Division, and, Rockwell and Draper Labs
- software A and B both compute all critical functions, such as ascent to orbit, descent from orbit, re-entry, etc.
- software A also does non-critical functions such as data collection
- significant life-cycle management applied to both software A and B
- software awarded “level 5” by CMU Software Engineering Institute
Recovery Block

- dynamic redundancy approach consisting of:
  
  o *primary routine* – executes critical software functions

  o *acceptance test* – tests output of primary routine after each execution & is considered “complete” if it catches all associated faults

  o *alternate routine* – performs same function as primary routine but may be slower or less capable, and is invoked by failure in acceptance test

- basic structure give as:

  Ensure T
  By P
  Else Q
  Else Error

  where T is acceptance test, P is primary routine, Q the alternate routine

- modifications have multiple versions of the alternate

- differences from N-version programming:

  o only single implementation of program run at a time

  o acceptability of results from test rather than comparison of functionally equivalent versions

  o in Recovery Block, P & Q deliberately designed to be as uncorrelated as possible

- in real-time applications, Q can be invoked if P not able to provide acceptable result within allocated time

- if acceptance test rejects results of Q, then is an abort condition

- can partition acceptance test into several separate tests, with first one(s) dealing with input

- typically P is more “desirable” in some sense, for example, time

- recovery blocks can be nested
- biggest drawback of technique is dependence on acceptance test for detecting any errors since no general rules for creating them

- big advantage is that can be applied to defined critical sections of code rather than to the whole system, which is much more economical

- problematic for real-time applications if have to execute all alternatives in worst case situation

**Composite Designs**

- use both techniques in same design

- with N-version programming can add data tests to begin corrective actions quickly

- in Recovery Block, can use multiple versions as acceptance test in short routines

**Distributed Recovery Block (DRB)**

- primary and alternate routines replicated and resident on two or more nodes interconnected by network

- in DRB refer to as *try block* and *alternate try block*

- in fault-free conditions, primary node runs primary try block and back-up node runs alternate try block concurrently

- if both pass acceptance test, update own local database

- on success of primary node, informs its backup which updates its own db with own result

- however, only primary outputs sent on to successor nodes

- if primary try block fails, primary attempts to inform backup which then takes over as primary node

- if primary has completely failed, can’t inform backup, but backup takes over after a time-out period
- if primary node is considered failed from error in try block, node tries to become backup so it can take over later if backup fails
- for such to happen, need checkpoints and rollback capability

**Acceptance Tests**

- two levels of acceptance tests:
  - **higher-level acceptance test** – tests that outputs of program consistent with functional requirements, also called *functional acceptance test*
  - **lower-level acceptance test** – tests that key variables and functions properly executed, also called *structural acceptance test*

- typically acceptance tests have following categories, which may overlap:
  - **satisfaction of requirements**
    - for small sections of code, e.g. when sorting numbers test if result uniformly descending, same number of elements as started with, etc.
  - **accounting tests**
    - used for transaction-oriented applications with simple math operations, e.g. airline reservations
  - **reasonableness tests**
    - pre-computed ranges, expected sequences of steps dependent on physical constraints, e.g. aircraft range of speeds
  - **computer run-time tests**
    - provided by most of today’s computers, e.g. divide by zero, overflow, underflow, undefined opcode, etc.

- acceptance tests either check for what program should or should not do
- no methodology for deciding what is best type of test
- considered *complete* if catches all associated faults
Fault Trees

- no method to anticipate all possible problems
- tend to have some sense of general classes of failures
- fault trees used to identify such classes in top-down method
- fault trees help in review of FT for software, and used in hardware FT
- basic concept is that top event (i.e. failure of system) can be divided into simpler events linked together by AND or OR gates
- continue break-down until have units with known probability of failure
- then work backward up the tree
- given event covered if all events leading into OR gate can be tested, or any one of the events leading into AND gate can be tested (i.e. all must fail at same time for failure)
- if all higher-level events can be tested by acceptance tests, system has 100% coverage
- doesn't guarantee comprehensive coverage but helps in placement of acceptance tests in a more pictorial method

Rollback and Recovery

- basic technique is to back up the computation to the last set of valid data and continue computation from there assuming that the error is not hard but rather a transient one
- an example would be if you go to print something from the desktop and the printer is turned off; are sent back to control and you can turn on the printer and re-send the command to print
- transient software error is due to software fault for particular software state
- if repeat computation again but software state has changed, then good probability error will not recur
two types of recovery techniques:

- *forward error recovery* – continue with operation knowing error exists and will attempt to correct later, e.g. tracking in air-traffic control

- *backward error recovery* – restart or rollback computation process to some point before occurrence of error and restart computation

- discuss below several versions of backward error recovery

**Rebooting**

- simplest but weakest approach is to reboot

- users of home or small businesses familiar with this approach which for word-processing, email, business files, can be problematic is frequent saves are not done

**Journaling Techniques**

- more complex and somewhat quicker than reboot/restart since only subset of inputs must be saved

- basic approach is:
  
  - copy of original database, disk, and filename to be stored
  
  - all transactions (inputs) that affect data must be stored during execution
  
  - process backed up to beginning and computation retried

- only can be executed for given time period, then erase and start new journaling period

- need to balance the amount of storage required vs. processing time for computational retry
Retry Techniques

- these techniques are faster than the above, but more complex since more redundant process-state information must be stored
- retry happens immediately after an error is detected
- if transient error, only wait for it to die out, and then retry begins
- if hard error, then reconfigure system, followed by retry
- for both types of errors, must keep the whole system state in storage
- if data has been modified and cannot be restored, will fail

Checkpointing

- only uses software, while retry can require dedicated hardware also
- retry saves entire time history of system state during period, but with checkpointing time history of system state is saved only at specific points so less storage needed
- disadvantage of checkpointing is amount and difficulty of programming associated
- on failure, go back to last checkpoint and system state set to stored values and restarted
- if successfully restored, process continues, and have just lost time for restoration and any new input data
- if not successful, can rollback to previous checkpoint
- if error has irrevocably modified some data, checkpoint approach fails
- good example is Tandem’s Guardian operating system