Multi-version approach (with error detection and recovery)

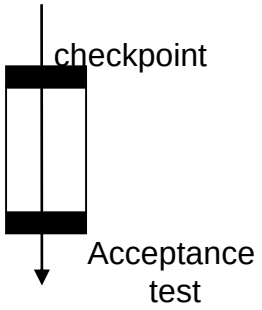
Backward error recovery: Recovery block

Brian Randell early 1970s at Newcastle

- Accettability of the result is decided by an acceptance test **T**
- Primary alternate, secondary alternates

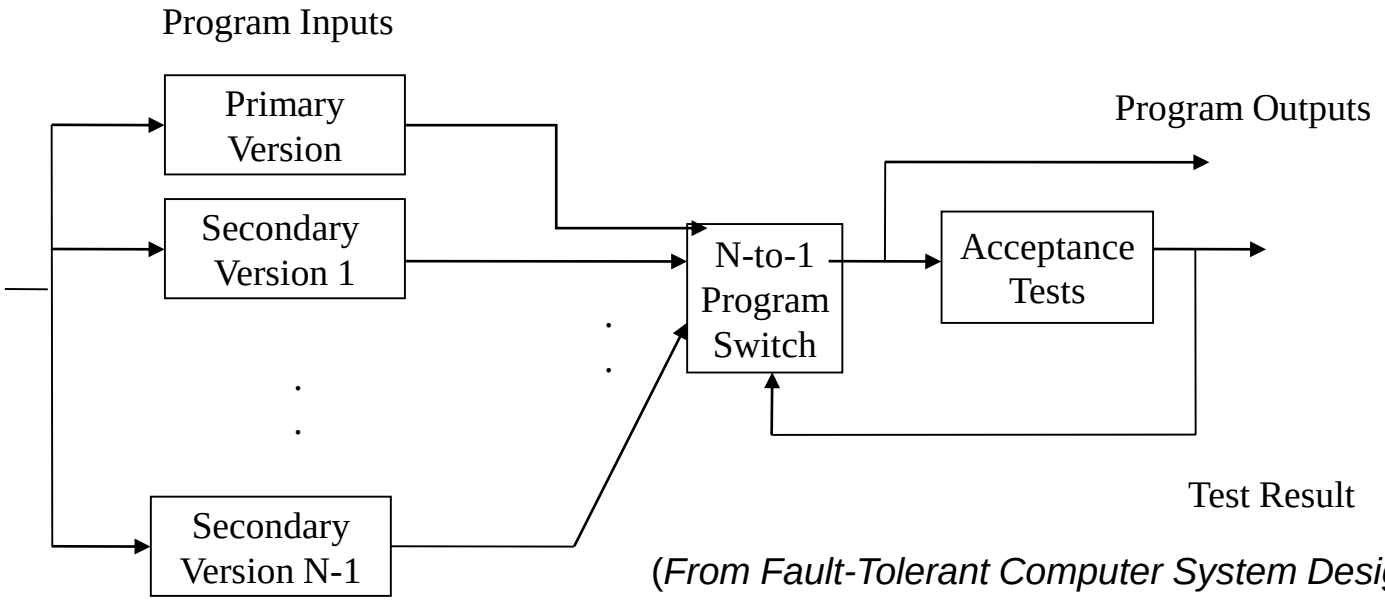
Basic structure:

Ensure T
By P
else by Q
Else error



- Each recovery block contains variables global to the block that will be automatically checkpointed if they are altered within the block.
- Upon entry to a recovery block, the primary alternate is executed and subjected to an acceptance test to detect any error in the result.
  If the test is passed, the block is exited.
  If the test is failed or the alternative fails to execute, the content of the recovery cache pertinent to the block is reinstated, and the second alternate is executed.
  This cycle is executed until either an alternative is successful or no more alternatives exist. In this case an error is reported.

Backward error recovery: Recovery block



(*From Fault-Tolerant Computer System Design D. K. Pradhan, Prentice Hall, 1996)*

- A single acceptance test
- Only one single implementation of the program is run at a time
- Combines elements of checkpointing and backup
- Minimizes the information to be backed up
- Releases the programmer from determining which variables should be checkpointed and when
- linguistic structure for recovery blocks requires a suitable mechanism for providing automatic backward error recovery. Randell produced the first such "recovery cache“ scheme

Recovery block in concurrent systems

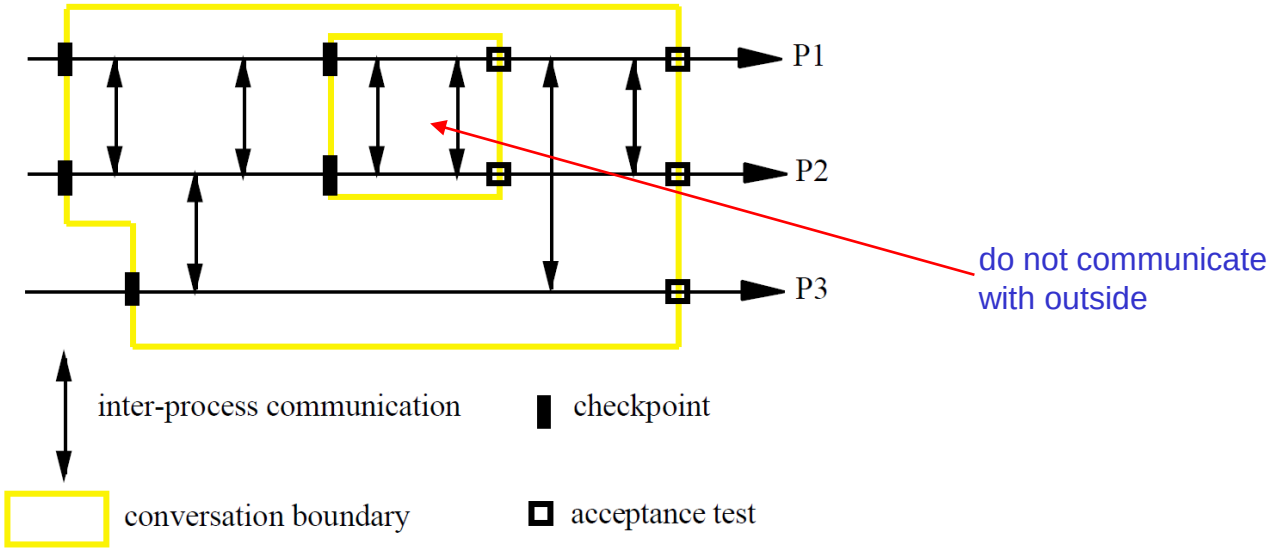
When a system of cooperating processes employs recovery blocks, each process will be continually establishing and discarding checkpoints, and may also need to restore to a previously established checkpoint.

However, if recovery and communication operations are not performed in a coordinated fashion, then the rollback of a process can result in a cascade of rollbacks that could push all the processes back to their beginnings — the domino

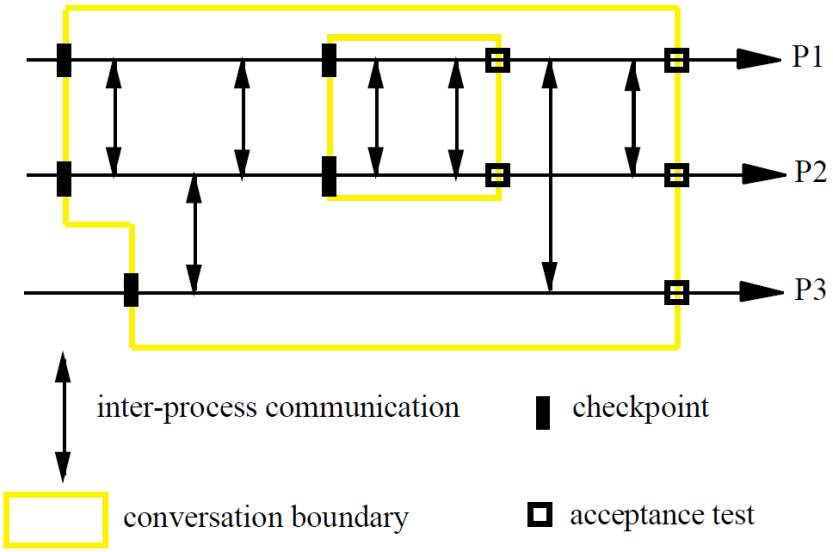
[Randell 1975] had come up with the notion of a *conversation* — something which we later realized was a special case of a nested atomic action.

Conversion scheme

- one of the fundamental approaches to structured design of fault-tolerant concurrent programs.
- provides a means of coordinating the recovery blocks of interacting processes



Example where three processes P1, p2 and P3 communicate within a conversation and the processes P1 and P2 communicate within a nested conversation



The operation of a conversation is: (i) on entry to a conversation a process establishes a checkpoint; (ii) if an error is detected by any process then all the participating processes must restore their checkpoints; (iii) after restoration all processes use their next alternates; and (iv) all processes leave the conversation together.

Real-time applications may suffer from the possibility of *deserters* in a conversation — if a deadline is to be met then a process that fails to reach its acceptance test could cause all the processes in the conversation to miss that deadline

Single-version software fault tolerance techniques

(redundancy applied to a single version of software to detect errors and recover)

Software faults

Bohrbugs
permanent design faults, deterministic in nature
identified during the testing and debugging phase

Heisenbugs
temporary internal faults (intermittent faults)
They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible.
For example faults at boundaries between various software components with timinig dependences.
They are state dependent and input dependent faults. (extremely difficult to identify through testing)

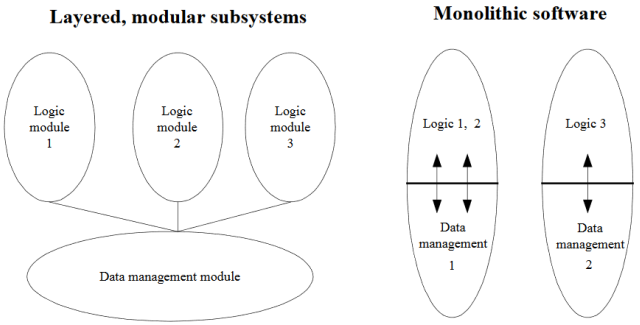
Basis principles to implement fault tolerance

- modular software architecture
- system closure principle
- self-checking and self-protection principle

Modular software architecture

"Modular software architecture helps us view the system, not just in layers or services, but as composition of small modules."

"A software module is a deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers."



Best practices:

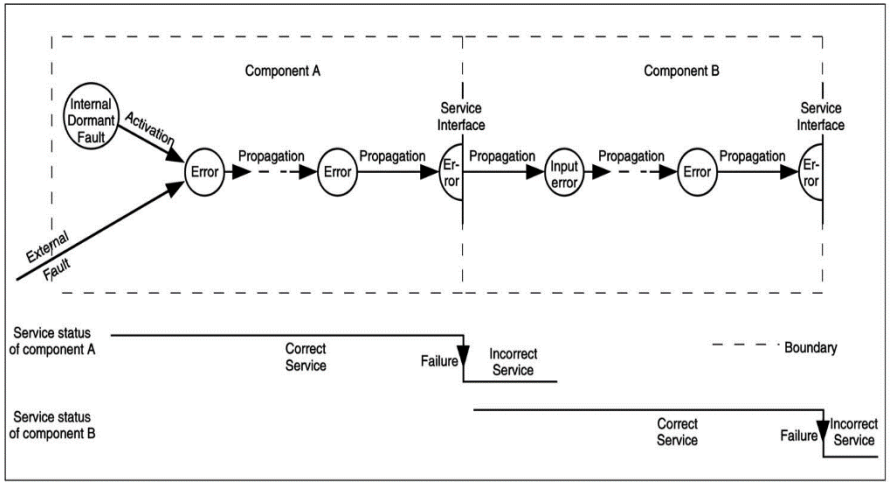
1) add *error detection* capability to modules

2) use the hierarchy and connectivity of modules to analyse *error propagation*

3) partitioning of module into
   - functional independent modules
   - control modules (that coordinate the execution)

   and provide isolation between functionally independent modules for *error confinement*

4) structuring of the activity between interacting components into *atomic actions*



Error dectection and recovery in programs:
Exception handlers

Atomic action:

activity in which the components interact with each other and there is no interaction with the rest of the system for the duration of the activity

Atomic action: provides a framework for error confinement and recovery (if a failure is detected during an atomic action, only the participating components can be affetcted)

Example: transactions in databases

System closure principle

no action is permissible unless explicitly authorized (mutual suspicion)

1. Each component is only granted the capabilities needed to execute its function

2. Each component examines each request or data item from other components before acting on it
   For example, each software module checks legality and reasonableness of each request received

3. A capability disabled by an error does not result in an undesirable action, only disables a valid action

Self-protection and self-checking principles

Software system: a set of communicating components

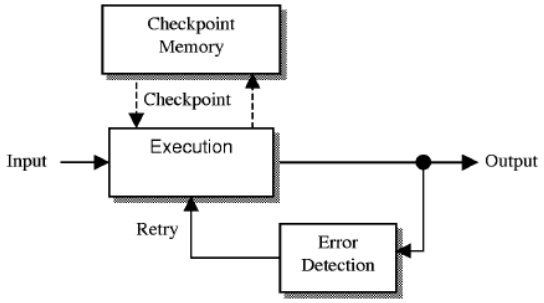
Component (self-protection): protect itself by detecting errors in the information received by other interacting components

Component (self-checking): able to detect internal errors and take appropriate actions to prevent the propagation to other components

Checkpointing and restart recovery mechanism

Most of the faults at this stage are *Heisenbugs,* hence these faults result in transient failures, i.e., failures which may not recur if the software is restarted.

Restart is usually enough to successful completion of the execution of the module



W. Torres-Pomales
Software fault tolerance: A tutorial
NASA,/TM-2000-210616, 2000

Checkpointing and restart recovery mechanism

- **Static**
  restart from predetermined states
  (initial state or intermediate state, ..)
- **Dynamic**
  restart from checkpoints created during
  the execution of the module (backword error recovery)

Error detection checks

Reasonableness checks: use known sematic properties of data (acceptable range of variables, rate of change, acceptable transitions, probable results…) Based on the design requirements of a module.

Reversal checks: inverse computation use the output to compute the corresponding inputs

assume the specified function of the system is to compute a mathemathical function, output = F(input) if the function has an inverse function F', such that F'(F(x))=x, we can compute F'(output) and verify that F'(output) = input

Coding checks: use coding in the representation of information technique developed for hardware can be used for software (the content of the data is not changed)

Error detection checks

Structural checks: use known properties of data structures lists, trees, queues can be inspected for a number of elements (redundant data structure could be added, extra pointers, embedded counts, …)

Timing checks: watchdog timers check deviations from the acceptable module behaviour

Run-time checks:

error detection mechanism provided in hardware (dived by 0, overflow, underflow, …) can be used to detect design errors

Error Recovery

Forward recovery transform the erroneous state in a new state from which the system can operate

Backward recovery bring the system back to a state prior to the error occurrence - Checkpointing

Backward and forward recovery are not exclusive they can be combined if the error persists

Forward error recovery

Requires to assess the damage caused by the detected error or by errors propagated before detection

Usually ad hoc

Example of application:

real-time control systems, an occasional missed response to a sensor input is tolerable

The system can recover by skipping its response to the missed sensor input.

Backward error recovery: Checkpointing

A copy of the current state for possible use in rollback is called checkpoint.

Checkpoints

- may be taken automatically (periodically) or upon request by program
- need to be correct
- need eventually to be discarded
- survival of checkpoint data

Backward error recovery: Checkpointing

Checkpointing/rollback (resetting the system and process state to the state stored at the latest checkpoint) need mechanisms in run-time support

Organisation of fault tolerance

From A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, Vol. 1, N. 1, 2004