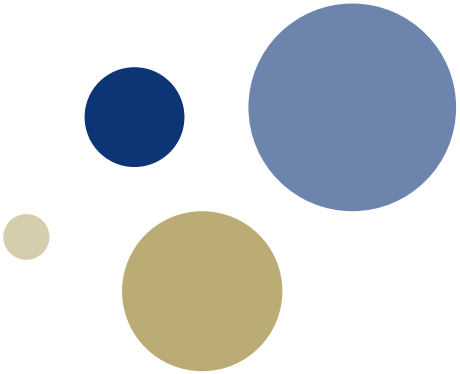
NTNU – Trondheim
Norwegian University of
Science and Technology



TTK4145 – Real-time Programming

Lecture 8 – Fault Tolerance

Example exam questions

- From 2016:
  - **1-1)** The term “Fault Tolerance” encompasses more/something else than just minimizing the number of bugs in the system. What?
  - **1-2)** Testing your system is seen as an insufficient technique for making a fault tolerant system. Why?

- From 2014:
  - **5-1)** Acceptance tests is seen as an important tool for handling errors. What do we gain from using acceptance tests in addition to the more traditional tests on error conditions?
  - **5-2)** Give examples of what one can test for when making acceptance tests

- From 2015:
  - **4-3)** The failure modes is the ways a system can fail. To “merge failure modes” is a technique: What do we gain by doing this ?

Learning goals: Fault Tolerance

- Understand and use terms (like): Reliability. Failure vs fault vs error. Failure modes. Acceptance test. Fault prevention vs. tolerance. Redundancy, Static vs. Dynamic. Forward/ Backward error recovery.

- Understand, use and evaluate techniques (like): N-version programming. Recovery blocks. Error detection. Failure mode merging. Acceptance tests

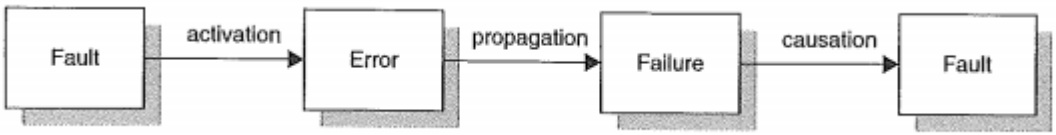
What is a fault?

Whose fault is it anyway?

Reliability, failure and faults

- Reliability:
  - a measure of the success with which the system conforms to some authoritative specification of its behaviour.
- Failure
  - When the external behaviour of a system deviates from that which is specified for it
- Error
  - Unexpected problems in the system internals that manifest themselves in the system's external behaviour
- Fault
  - The mechanical or algorithmic cause of an error
  - Active when an error is produced, otherwise dormant

- A system usually consists of components, meaning smaller subsystems
  - Failure in one component (subsystem) will lead to fault in another (supersystem)
  - Causes chain reaction, known as fault-error-failure-fault chain



Fault durations

- Transient faults
  - occurs at a particular time, remains in the system for some period and then disappears (but not necessarily the induced error).
  - Initially dormant, but can activate at any time
  - Example: hardware components which have an adverse reaction to some external interference, such as electrical fields or radioactivity.
- Permanent faults
  - start at a particular time and remain until they are repaired
  - Example: a broken wire or a software design error.
- Intermittent faults
  - transient faults that occur from time to time.
  - Example: heat sensitive components

Software faults

- Also known as bugs
  - Bohrbugs: reproducible and identified
  - Heisenbugs: occur in rare conditions, usually disappear upon inspection
- Statistics from IBM
  - 1 bug / 50 lines of code before testing
  - 1 bug / 500 lines of code at release
  - 1 bug / 550 lines of code after a year, then constant
- 20 % of a project size is dedicated to maintenance. Does software deteriorate with age?
  - Unlike hardware, written code never change
  - However, faults can remain dormant for a long time
    - Normally related to resources, memory leaks are a good example
    - Become active after significant continual use of software
    - Usually solved by restarting the system

Traditional error handling

- The “normal” way of handling bugs
  - Make your program to spec
  - Test the program
  - Errors happen
  - Find cause in code
    - Erroneous code
    - Missing handling of some situation
    - Incomplete spec
  - Add/Change code
    - Fix code
    - Detect/handle situation
- Example shows a typical end-result
  - One line of readable code grows into something unmanageable
  - Are the cases added during debugging?
  - How do we test and maintain the code?

```
FILE * openConfigFile(){
     FILE * f = fopen("/.config.cfg","r");
     if(f == NULL){
           switch(errno){
                 case ENOMEM: {
                       ...
                       break;
                 }
                 case ENOTDIR:
                 case EEXIST: {
                       // ERROR!
                       break;
                 }
                 case EACCESS:
                 case EISDIR: {
                       ...
                       break;
                 }
                 ....
           }
      }
      return f;
}
```

Failure modes for `fopen()` from `clib`

On error `NULL` is returned, and the global variable `errno` is set:

`[EINVAL]` The mode provided to `fopen` was invalid.

The `fopen` function may also fail and set `errno` for any of the errors specified for the routine `malloc`(3):

`[ENOMEM]` Memory allocation error.

The `fopen` function may also fail and set `errno` for any of the errors specified for the routine `open`(2).

`[EEXIST, EISDIR, EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, ENXIO, ENODEV, EROFS, ETXTBSY, EFAULT, ELOOP, ENOSPC, ENOMEM, EMFILE, ENFILE]`

Challenges with traditional error handling

- Failure modes destroy modules boundaries
  - Knowledge of the internals are required
- Testing only shows the presence of faults, not the absence
- Sometimes impossible to test under realistic conditions
- Almost impossible to test for race conditions (Heisenbugs)
- Faults are more often the result of unexpected situations then pure software bugs
  - Errors introduced at the requirements stage of the project might only manifest when the system goes operational
- Embedded systems may have higher demands for safety
  - Must ensure that the system stay safe despite of unexpected faults
- How can we handle faults still left in the system after testing, and ensure that also unexpected faults are handled?

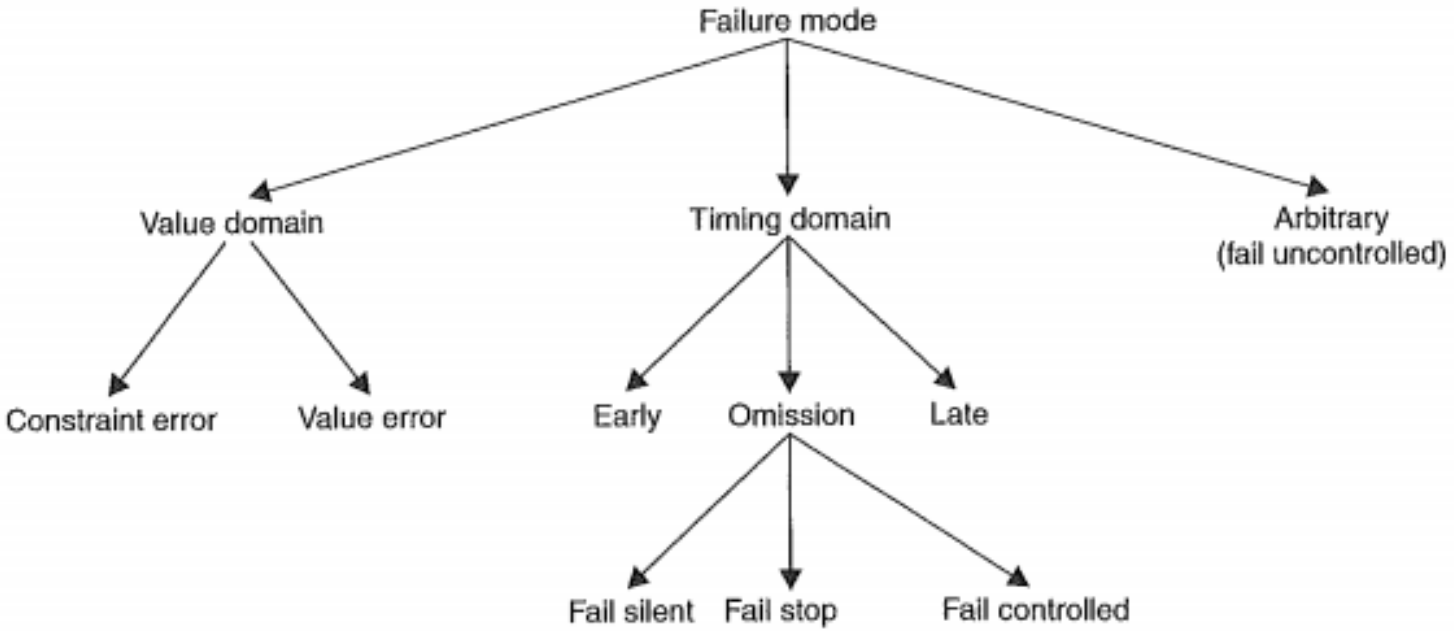
Fault tolerance

It doesn’t matter whose fault it was

Failure modes

- In which manners can the system fail?
  - Which situations do we need to handle?
- Value failure
  - The value associated with the service is in error
    - The value is outside the range expected from the service -Constraint error
    - Everything else – Value error
- Time failure
  - The service is delivered at the wrong time
    - too early — the service is delivered earlier than required
    - too late — the service is delivered later than required (often called a performance error);
    - Infinitely late — the service is never delivered (often called an omission failure).
- Arbitrary failure
  - Combination of the above

Classification of failure modes



Classification of failure modes

- Fail uncontrolled
  - a system which can produce arbitrary errors in both the value and the time domains (including impromptu errors).
- Fail late
  - a system which produces correct services in the value domain but may suffer from a ‘late’ timing error.
- Fail silent
  - a system which produces correct services in both the value and time domains until it fails; the only failure possible is an omission failure and when this occurs all following services will also suffer an omission failure.
- Fail stop
  - a system which has all the properties of fail silent, but also permits other systems to detect that it has entered the fail silent state.
- Fail controlled
  - a system which fails in a specified controlled manner.
- Fail never
  - a system which always produces correct services in both the value and the time domain.

Fault prevention

- Attempts to eliminate any possibility of faults in a system before it goes operational

- Fault avoidance
  - Rigorous specification of requirements
  - Use of proven design methodologies
  - Use of analysis tools to verify key program properties
  - Use of languages with facilities for data abstraction and modularity
  - Use of software engineering tools to manage complexity

- Fault removal
  - Reviews/Inspections
  - Verification
  - Testing

Failure of Fault Prevention Approach

- Already discussed why traditional testing fails
- Additionally, hardware components will still fail, meaning fault prevention fails when
  - either the frequency or duration of repair times are unacceptable

  or
  - the system is inaccessible for maintenance and repair activities
    - The crewless spacecraft Voyager is currently 10 billions miles from the sun
- Alternative is **Fault Tolerance**

Fault tolerance

- Enables a system to continue functioning even in the presence of faults
- Several different levels of fault tolerance can be provided by a system
  - **Full Fault Tolerance:** the system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance
  - **Graceful Degradation (fail soft):** the system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair
  - **Fail Safe:** the system maintains its integrity while accepting a temporary halt in its operation
- The level required will depend on the application
- Most safety critical systems require full fault tolerance, however in practice many settle for graceful degradation

Merging failure modes

- What is the worst that can happen?
  - We must handle this anyway, so group with easier ones (lots of these are known situations
- We achieve:
  - A simpler system overall
  - Simpler error handling.
  - Better modules.
  - Handling of unknown errors.
- Examples:
  - All communication errors: Lost message, resend
  - All SW bugs: restart.
  - All exceptions: ABORT/Try Again.

```
FILE * openConfigFile(){
    return fopen("/.config.cfg","r");
}

// The caller
void initialize(){
...
    FILE f = openConfigFile();
    struct SConfig * config =
           readConfiguration(f);

    // The acceptance test:
    if(!checkConfiguration(config)){
        // Fall back to default
        // config, warn user.
    }
...
}
```

Acceptance test

- A change of perspective from the traditional focus on
  - Error situations
  - Error returns
- Need to handle unexpected errors such that all failure modes are well-defined and handled
- The main philosophy is to put demands on the current status, is it reasonable?
  - Provides a form of error detection
- Note that the term used is acceptance not correctness
  - allows a component to provide a degraded service
- We will come back to the design of acceptance tests

Acceptance test for `openConfigFile()`

```
int checkConfiguration(struct SConfig * config)){
    // check range of all variables
    if(config->nNodes < 1 && config->nNodes > 5) return 0;
    ...
    // Check any correlations
    if(config->maxOrders > config->nNodes * 4) return 0;
    ...
    // Possibly configuration have been extended with
    // parameters enabling better checks
    if(config->isSimpleSystem && config->nNodes > 4) return 0;
    ...
    ...
    return 1;
}
```

Redundancy

Redundancy

Protective redundancy

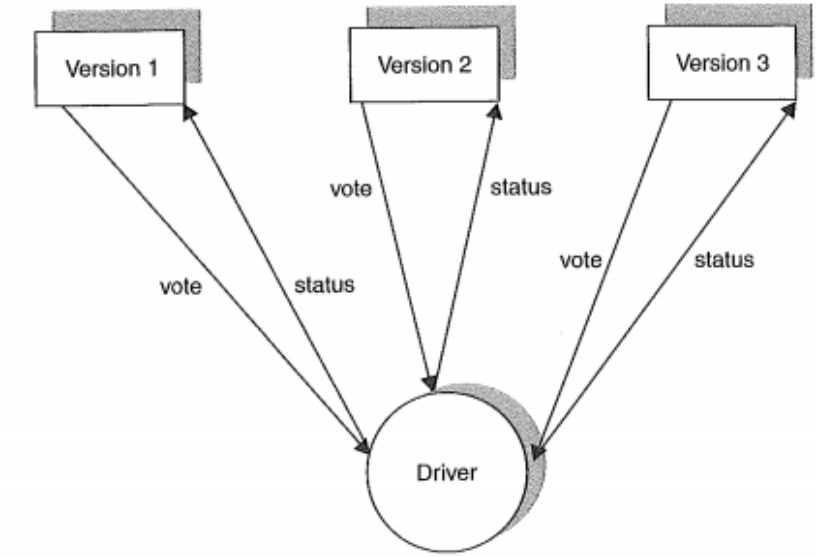
- Fault tolerant techniques rely on redundancy
  - Extra elements introduced to detect and recover from faults
- Redundant in the sense not required for a system's normal mode of operation
- Want to minimize redundancy while maximizing reliability
  - Requires more hardware/software, more complexity, less reliability, more faults
- Classifications of redundancy
  - Static redundancy
    - Also known as masking, aims to hide the effects of faults using redundant components within a (sub)system
    - Example: Triple (N) Modular Redundancy in hardware:
      - Three (N) identical subcomponents and majority voting circuit
      - Assumed fault is transient or due to component deterioration
  - Dynamic redundancy
    - Indicates explicitly or implicitly that the output is in error
    - Recovery must be handled by separate component
    - Error detection rather than error masking
    - Example: Checksums on communication, parity bits in memory

N-Version programming

- Defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same specification
  - *N* software teams produce the *N* versions of the software without interaction (design diversity)
- The programs execute concurrently with identical inputs, and results (votes) compared by a driver process
- Assumptions
  - A program can be completely, unambiguously and consistently specified
  - Programs developed independently will fail independently
- Last assumption requires each version written in a different programming language and different development environments
  - Alternatively, use compilers and support environments from different manufacturers
- The N versions should be distributed to separate machines with fault tolerant communication lines

Driver process

- The driver process is responsible for
  - Invoking each version
  - Waiting for completion
  - Comparing and acting on results
- The *N* versions must interact with the driver
  - Comparison vector: Output data
  - Comparison status indicator: Continue/terminate
  - Comparison points: When votes are submitted

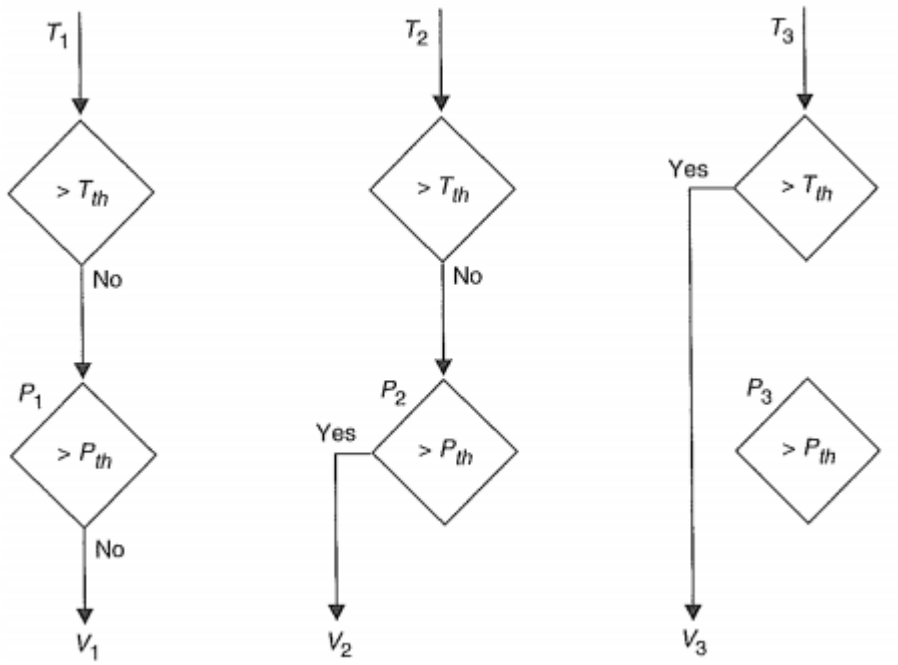


Challenges

- More parts increase complexity
  - Higher chance of faults
  - Maintainability decreases.
- The driver process must be fault tolerant
  - N-Version drivers only propagate the problem
- The cost of the project increases N-fold
  - Software is the most expensive part of any embedded project
- Ensure truly independent versions
  - Common specification
  - Common programming culture
  - Common solutions to difficult aspects
- Specification errors will manifest in all N versions
  - The great majority of software faults stem from specification errors
- Voting comparisons for results with real numbers
  - Inexact voting: can conduct a range check, but what is the range?
  - Consistent comparison problem

Consistent comparison problem

- Problem occurs when reaching a threshold is basis for vote
- Inexact comparison with tolerance only shifts the problem to the tolerance borders
- Example: Taking corrective action based on temperature and pressure readings



Dynamic redundancy

- Redundant components only come into operation when an error has been detected

- The technique has four constituent phases:
  1. Error detection
  2. Damage confinement and assessment
  3. Error recovery
  4. Fault treatment and continued service

Error detection

- Replication checks
  - Two-version redundancy (N-Version programming)
- Timing checks
  - Watchdog timer
  - Only checks timing, not correct function of component
- Reversal checks
  - Feasible for isomorphic (one-to-one) relations between input and output
  - Calculates what input should be based on output, and compares with actual input
  - Square root of input, check by squaring output and comparing
- Coding checks
  - Tests for corruption of data
  - Checksum
- Reasonableness checks
  - Knowledge of the internal design and construction of the system
  - Check if state of data or value of an object is reasonable
- Structural checks
  - Check integrity of data objects such as lists and queues
  - Number of elements, redundant pointers or extra status information
- Dynamic reasonableness checks
  - Relationship between two time-adjacent outputs
  - Error can be assumed if the values differ greatly

Constituent phases of dynamic redundancy

1. Error detection
2. **Damage confinement and assessment**
3. Error recovery
4. Fault treatment and continued service

Damage confinement and assessment

- Usually delay between fault occurring and error being detected
  - Erroneous information may have spread throughout the system, must assess the damage done
- Damage assessment is closely related to damage confinement precautions taken
  - Ensure that the program is designed to answer the question:
    "What must be done to handle an error?"
- Damage confinement is concerned with structuring the system so as to minimise the damage caused by a faulty component (also known as firewalling)
  - Be careful not to spread unsafe data
  - Mark or store safe, consistent states (recovery points)
- Modular decomposition provides static damage confinement
  - Make modules with well-defined interfaces and hidden internals
  - Include internal consistency checks and error handling
  - Makes it difficult for errors to propagate between modules
- Atomic actions provides dynamic damage confinement
  - Actions that appear indivisible to the rest of the system
  - Also called transactions
  - No information can be passed from within the atomic action to the rest of the system
  - Moves the system from one consistent state to another

Constituent phases of dynamic redundancy

1. Error detection
2. Damage confinement and assessment
3. **Error recovery**
4. Fault treatment and continued service

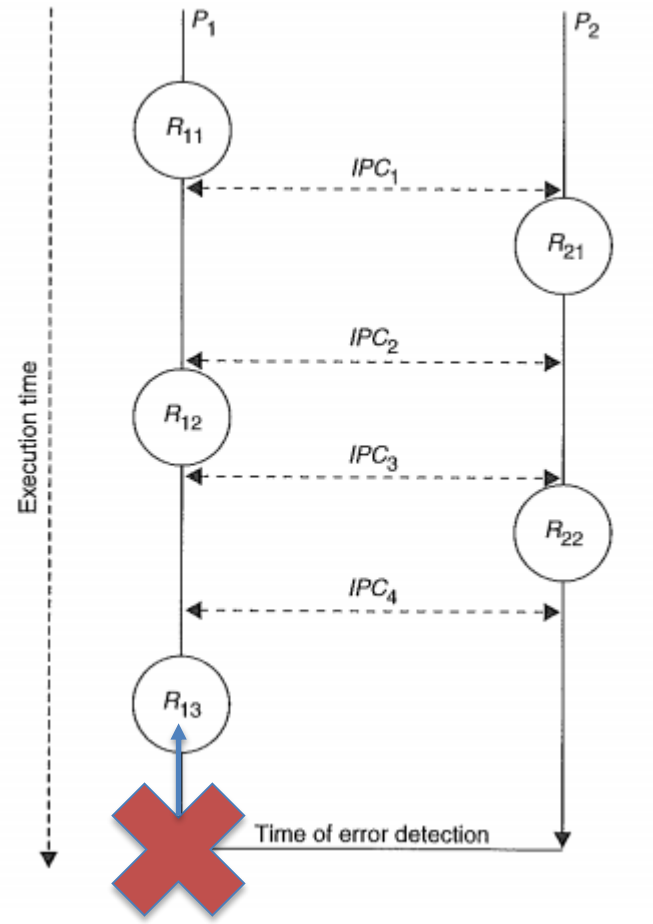
Error recovery

- The most important part of any fault-tolerance technique
- Two approaches: *forward* and *backward error recovery*
- *Forward error recovery*
  - attempts to continue from an erroneous state by making selective corrections to the system state to arrive at a consistent state
  - makes safe any aspect of the controlled environment which may be hazardous or damaged because of the failure
  - is system specific and depends on accurate predictions of the location and cause of errors (damage assessment)
  - Examples: redundant pointers in data structures and the use of self-correcting codes such as Hamming Codes

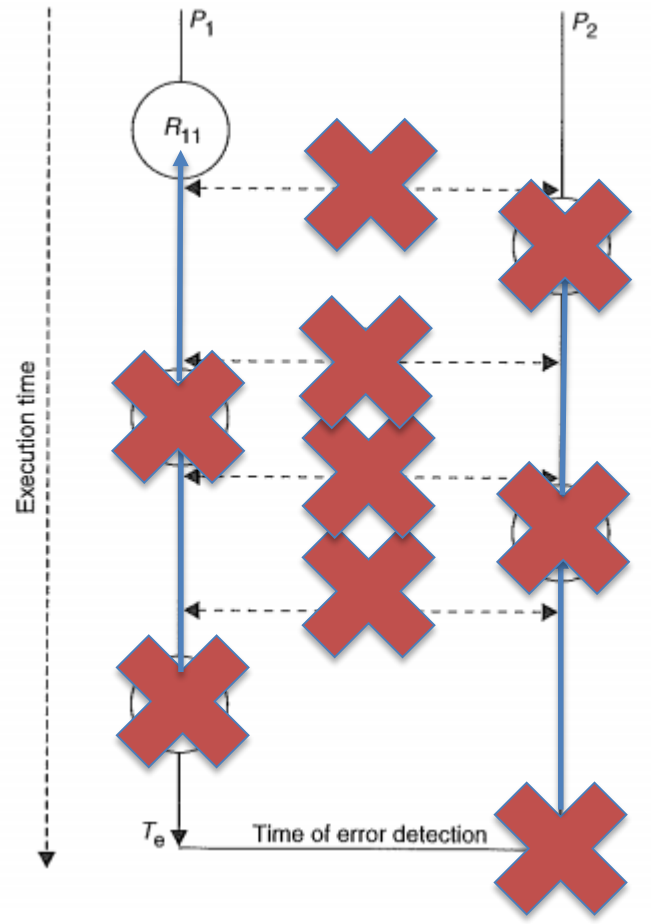
Backward error recovery

- Relies on restoring the system to a previous safe state and executing an alternative section of the program
- This has the same functionality as the fault-producing section, but uses a different algorithm
- The point to which a process is restored is called a *recovery point* and the act of establishing it is termed *checkpointing* (saving appropriate system state information at run-time)
- Advantages:
  - the erroneous state is cleared and it does not rely on finding the location or cause of the fault
  - can recover from unanticipated faults including design errors
- Disadvantages:
  - can not undo effects the fault had on the environment
    - difficult to unlaunch missiles
  - is time-consuming, might not be suitable for real-time applications
    - applications requiring sensor information might be time-dependent
  - encounters issues when concurrent processes interact
    - dependencies between recovery points

Domino effect



Error occurring in Process P_1 at time T_e: simple rollback to recovery point R_{13}.



Error occurring in Process P_2 at time T_e: domino effect due to Inter-Process Communication.

Constituent phases of dynamic redundancy

1. Error detection
2. Damage confinement and assessment
3. Error recovery
4. **Fault treatment and continued service**

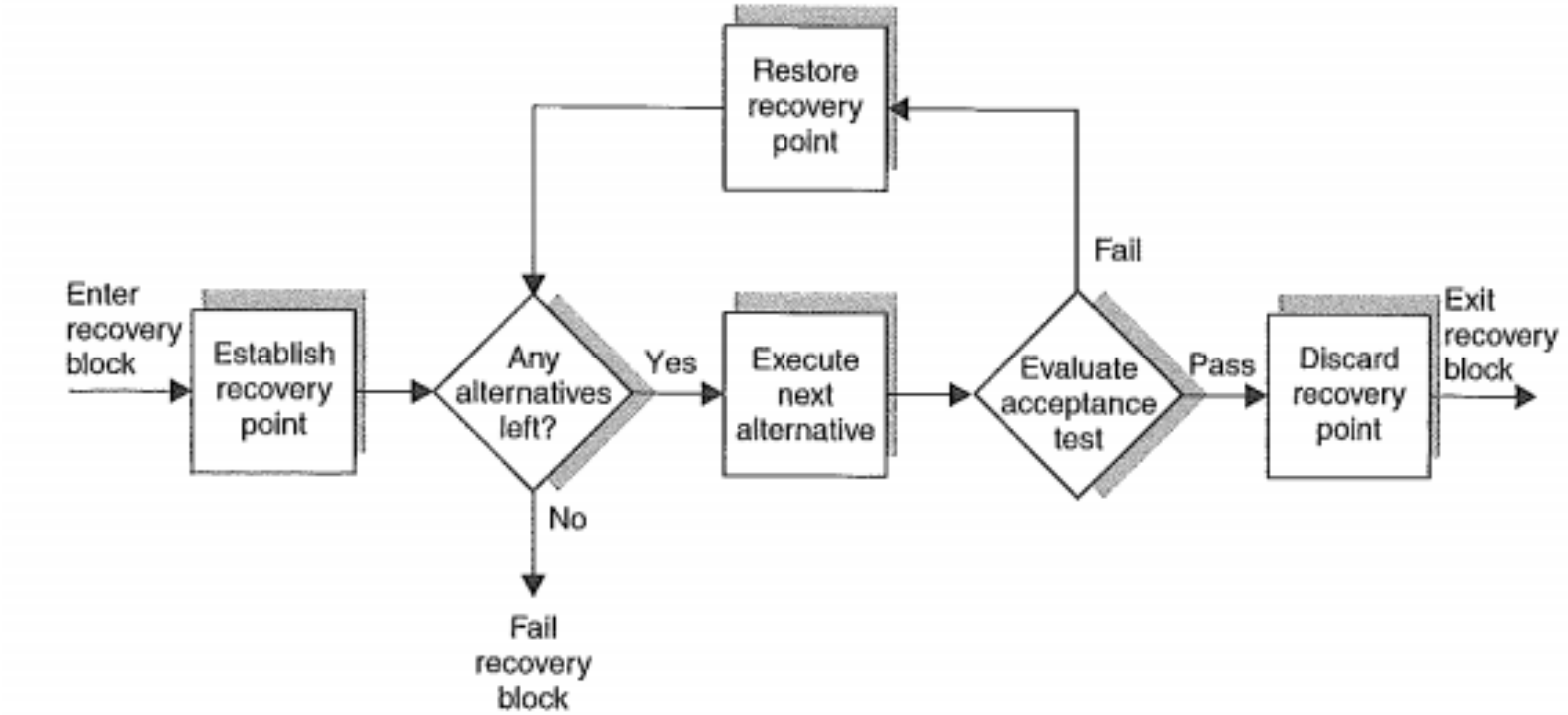
Fault treatment and continued service

- Error recovery returned the system to an error-free state, but the underlying fault might still be present
- The automatic treatment of faults is difficult and system specific
  - Some systems assume all faults are transient
  - others that error recovery techniques can cope with recurring faults
- Fault treatment can be divided into two stages:
  - Fault location:
    - Error detection techniques can help to trace the fault to a component.
  - System repair:
    - Hardware components can be replaced
    - Software faults can be removed in a new version of the code
      - In non-stop applications it will be necessary to modify the program while it is executing

Recovery blocks

- The dynamic redundancy counterpoint to N Version Programming
- At the entrance to a block is an *automatic recovery point* and at the exit *an acceptance test*
- The acceptance test is used to test that the system is in an acceptable state after the block's execution (primary module)
- If the acceptance test fails, the program is restored to the recovery point at the beginning of the block and an alternative module is executed
- If the alternative module also fails the acceptance test, the program is restored to the recovery point and yet another module is executed, and so on
- If all modules fail then the block fails and recovery must take place at a higher level

Recovery block mechanism



Recovery block syntax

- No commercially available real-time programming language has recovery blocks has a feature
- Suggested syntax:

```
ensure <acceptance test>
by
   <primary module>
else by
   <alternative module>
else by
   <alternative module>
   ...
else by
   <alternative module>
else error
```

```
ensure
Rounding_err_has_acceptab
le_tolerance
by
  Explicit Kutta Method
else by
  Implicit Kutta Method
else error
```