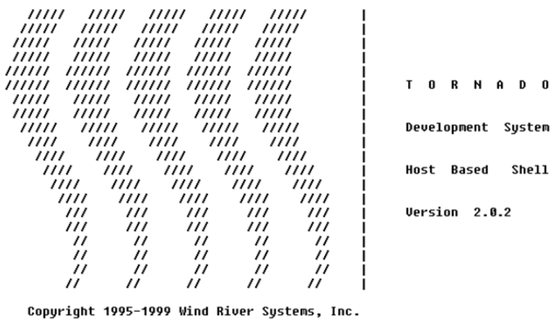
FYS 4220 / 9220 – 2011 / #9

Real Time and Embedded Data Systems and Computing

Reliability and Fault Tolerance – revised 30 Oct 2011



T.B. Skaali, Department of Physics, University of Oslo)

Murphy’s laws

„*Whatever can go wrong will do so, at the worst possible moment*”

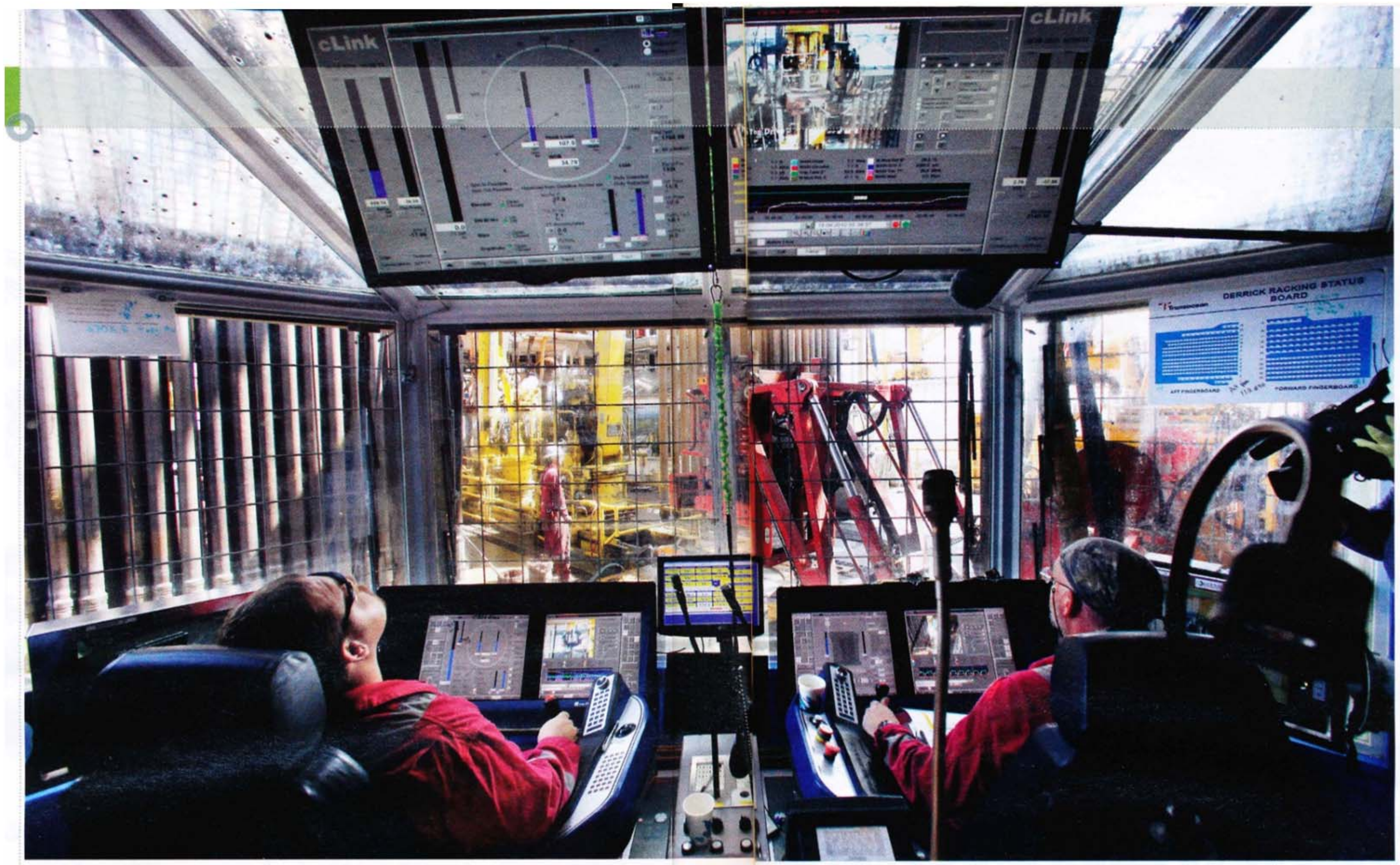
”If everything seems to be going well, you have obviously overlooked something„

Computer System Reliability and Nuclear War

http://www-ee.stanford.edu/~hellman/Breakthrough/book/chapters/borning.html

- On Tuesday, June 3, 1980, at 1:26 a.m., the display system at the command post of the Strategic Air Command (SAC) near Omaha, Nebraska, indicated that two submarine-launched ballistic missiles (SLBMs) were headed toward the United States. (1) Eighteen seconds later, the system showed an increased number of SLBM launches. SAC personnel called the North American Aerospace Defense Command (NORAD), who stated that they had no indication of attack.
- After a brief period, the SAC screens cleared. But, shortly thereafter, the warning display at SAC indicated that Soviet ICBMs had been launched toward the United States. Then the display at the National Military Command Center in the Pentagon showed that SLBMs had been launched. The SAC duty controller directed all alert crews to move to their B-52 bombers and to start their engines, so that the planes could take off quickly and not be destroyed on the ground by a nuclear attack. Land-based missile crews were put on a higher state of alert, and battle-control aircraft prepared for flight. In Hawaii, the airborne command post of the Pacific Command took off, ready to pass messages to US warships if necessary.
- Fortunately, there were a number of factors which made those involved in the assessment doubt that an actual attack was underway. Three minutes and twelve seconds into the alert, it was canceled. It was a false alert.
- NORAD left the system in the same configuration in the hope that the error would repeat itself. The mistake recurred three days later, on June 6 at 3:38 p.m., with SAC again receiving indications of an ICBM attack. Again, SAC crews were sent to their aircraft and ordered to start their engines.
- **The cause of these incidents was eventually traced to the failure of a single integrated circuit chip in a computer which was part of a communication system. To ensure that the communication system was working, it was constantly tested by sending filler messages which had the same form as attack messages, but with a zero filled in for the number of missiles detected. When the chip failed, the system started filling in random numbers for the "missiles detected" field.**

”IT systems full of errors”



Transoceans rigg full av datafeil

En faggruppe med norsk deltakelse har avdekket fatale feil på it-systemene på Deepwater Horizon før utblåsningen.

TEKST ESPEN ZACHARIASSEN
espen.zachariassen@tu.no

Datafrys og blå skjermer var hverdagen for boremannskapet på oljeriggen Deepwater Horizon i flere måneder før det smalt i våres.

Det kommer frem i forklaringer som mannskapet har gitt den amerikanske kystvakten og det amerikanske petroleumstilsynet, Bureau of Ocean Energy Management, Regulation and Enforcement. Teknisk Ukeblad har fått tilgang til utdrag av disse forklaringene.

En ekspertgruppe på risikohåndtering ledet av University of California, Berkeley, har analysert opplysningene og kommet frem til at omfattende feil og mangler ved it-systemene kan ha bidratt til ulykken.

Doktorgradsstipendiat i risikoanalyse Jon Espen Skogdalen ved Universitetet i Stavanger ble i juni tatt inn i denne ekspertgruppen.

Obama

Den gransker «Deepwater Horizon»-ulykken og rapporterer til Graham-Reilly Commission som er nedsatt av president Barak Obama.

En av USAs mest erfarne ulykkesgranskere, Dr. Robert Bea, leder gruppen på 50 forskere fra hele verden.

Bea har tidligere gransket redningsarbeidet etter orkanen Katrina, katastrofen med romfergen Colombia og over 20 ulykker på offshorerigger.

Fram til i dag har granskningsgruppen pekt på ulike tekniske feil bak ulykken i Mexicogulfen. Blant annet ufullstendig sementdesign og for dårlig kvalitetssikring og kvalitetskontroll i kritiske seksjoner av brønnen.

Gruppen har også kritisert prosedyrer og beslutninger hos operatøren og riggselskapet.

Kritisk

Nå har den i tillegg kommet fram til at sikkerhetskritiske it-systemer som ble brukt på Deepwater Horizon var fulle av feil.

Og feilene ble ikke forstått av mannskapet.

Ekspertgruppen presenterte sine funn for amerikanske Ptil og etatens direktør Michael Bromwich fredag.

Her trakk Skogdalen frem flere tilfeller av datasvikt som kan ha bidratt til ulykken:

- Boreoperatørens skjermer gikk i blått. Dette skjedde så ofte at fenomenet fikk betegnelsen «screen of death».

– Programvare stoppet opp til stadighet, slik at datamaskinen frøs.

– Mannskapet hadde manglende kontroll med utkoblinger og såkalte overbroinger av brann- og gassvarslingssystemet.

– Ved et tilfelle opplevde mannskapet såkalte «kick» eller «brønnspark» på grunn av datasvikt.

Et brønnspark innebærer en ustabilitet i brønnen som følge av at den tar inn gass, olje eller vann. Dersom barrierene svikter, kan brønnspark føre til utblåsning.

Avhengig av it

Deepwater Horizon var en såkalt femtegenerasjonsrigg, det vil si topp moderne og helt avhen-

The operators' screens went in blue so often that it was called "screen of death"

The programs often hung such that the system froze

Reliability and fault tolerance

- Factors influencing the reliability of a computer system, particularly critical for Real-Time / embedded:
  - Hardware level
  - Software level
  - System level
- Challenge:how can a system operate in the presence of errors?
  - How can one build systems with extreme reliability?
    - Or more relevant: with sufficient reliability!

Some of the presentation has been borrowed (thanks!) from the book of (York University computer science professors) Burns & Wellings, Ch. 5

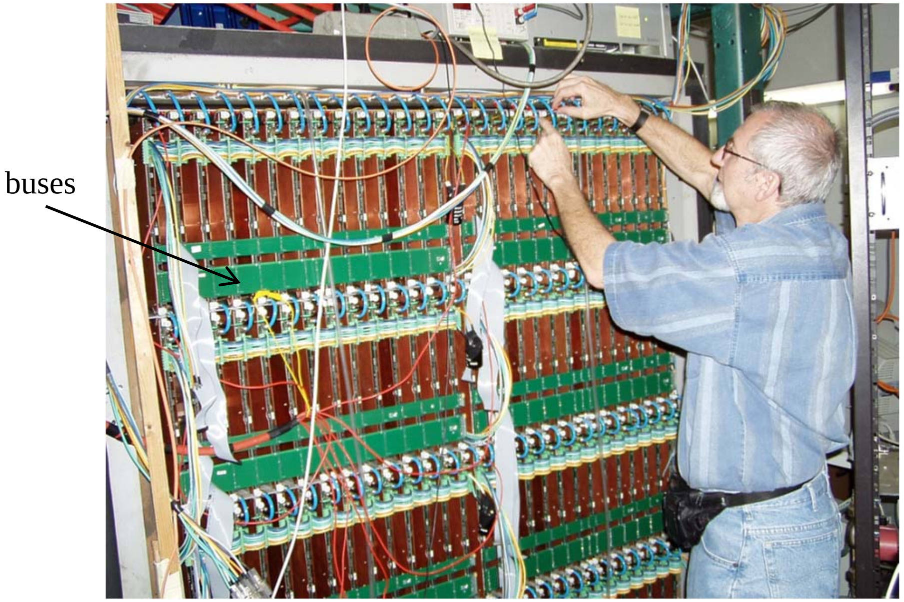
Chip and board failures

- Chip functional/logical faults
  – How can one test a chip with million of transistor equivalents?
  – Standard components like processor, memory and FPGA chips are tested (well, up to a point) by the manufacturer
  – However, many Real-Time / embedded systems are based on ”non-standard” Application Specific Integrated Circuits (ASICs)
    - As an example, the CERN detector electronics contains many ASICs. How can they be 100% tested? They can not, one just have to get around the faults, but first one has to detect them
- Board failures
  – Production issues: mounting (soldering, bounding)
  – Cooling related problems
  – Power problems, connectors
- Special cases:
  – Radiation tolerant – radiation hard,
  – Extreme environment: temperature, humidity etc

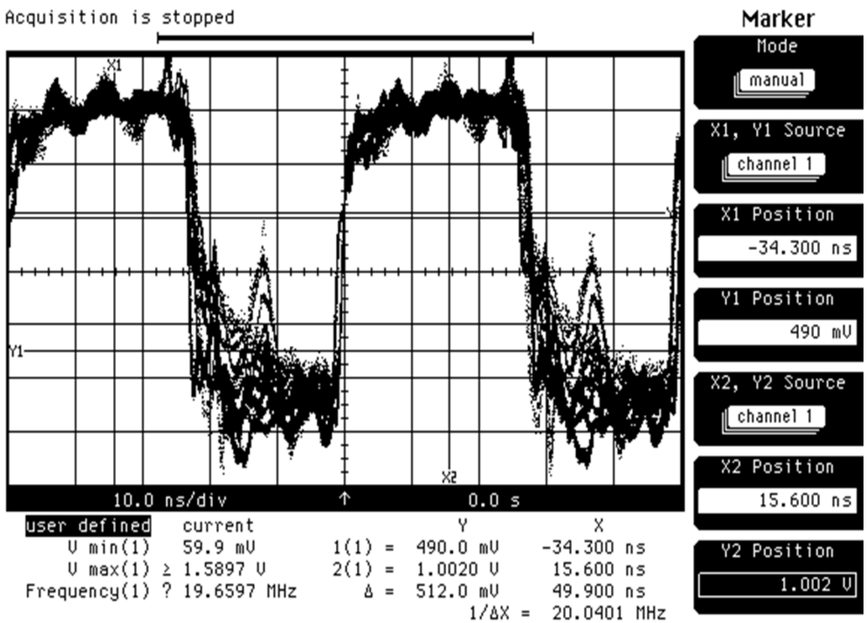
Hardware related failures

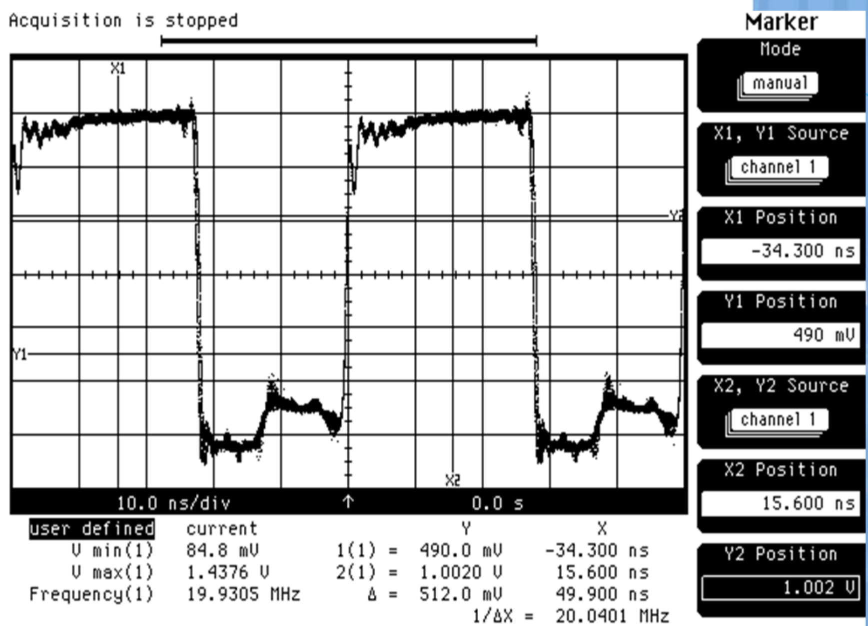
- Failures in electronic systems can have many reasons, and a systematic study is far outside the scope of this course. Some examples:
  - Intrinsic failures in a basic circuit (gate), for instance due to radiation damage (the probability for this increases with altitude since the radiation goes up!)
  - Interconnection failures
  - Connector and cable weaknesses
  - Power problems
  - Aging
  - Sloppy design
  - Plus all others …..
- Keep in mind: engineers are never wrong; they just make bad assumptions!

A case from a CERN experiment



- The picture to the left shows digitizer cards for 7128 channels. The data is read out over busses implemented as PCB strip lines.
- We had problems with corrupted data. The left scope trace shows reflections and cross-talk on the bus clock line, some spikes even cross the reference level. The right scope trace shows the improved clock signal with better impedance matching.





The mythical Mean Time Between Failure (MTBF)

- MTBF gives an estimate of the expected lifetime of a component
- Calculation of MTBF – methodology
  - A prediction process thereby a numerical estimate is made of the ability, with respect to failure, of a design to perform its intended function. Once the failure rate is determined, MTBF is calculated as the inverse of the failure rate
  - $MTBF = 1/(FR_1 + FR_2 + FR_3 + \ldots\ldots + FR_n)$ where FR_i is the failure rate of each component of the system
  - The failure rate is dependent on the operating environment
  - What you get is what you pay for! For instance, radiation tolerant and in particular radiation hard components are much more expensive than off-the-shelf stuff
  - The baseline is that any component will eventually fail!
- Web MTBF calculations: http://www.sqconline.com/reliability

Reliability and Fault Tolerance

Goals

- To understand *(some of)* the factors influencing the reliability of a hardware system
- To understand *(some of)* the factors which affect the reliability of a system and how software design faults can be tolerated.

Topics

- Reliability, failure and faults
- Failure modes
- Fault prevention and fault tolerance
- N-Version programming
- Software dynamic redundancy
- The recovery block approach to software fault tolerance
- A comparison between N-version programming and recovery blocks
- Dynamic redundancy and exceptions
- Safety, reliability and dependability

Scope

Four sources of faults which can result in system failure:

- **Inadequate specification**
- **Design errors in software**
- **Processor or component failure**
- **Interference on the communication subsystem**
- In this chapter *(ref B&W)*, some of the general design techniques that can be used to improve the overall reliability of embedded computer systems are considered.
- Exception handling, intimately connected with fault handling

Reliability, Failure and Faults

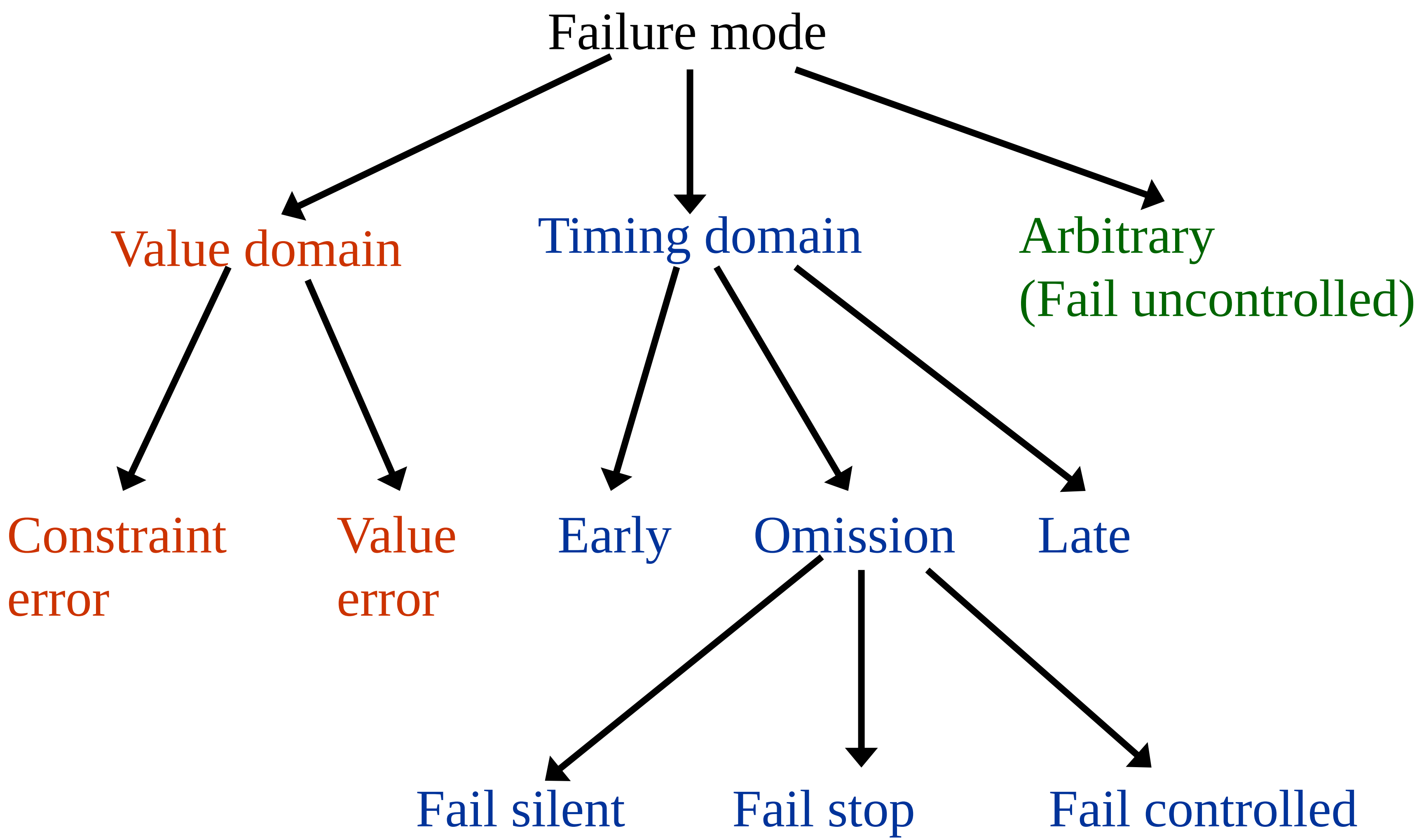
- The reliability of a system is "a measure of the success with which it conforms to some authoritative specification of its behaviour". *(Definition from 1978!)*
- When the behaviour of a system deviates from that which is specified for it, this is called a **failure**
- Failures result from unexpected problems internal to the system which eventually manifest themselves in the system's external behaviour
- These problems are called **errors** and their mechanical or algorithmic cause are termed faults
- Systems are composed of components which are themselves systems: hence the chain

→ failure → fault → error → failure → fault

Fault Types

- A **transient fault** starts at a particular time, remains in the system for some period and then disappears (well..?)
  - E.g. hardware components which have an adverse reaction to radioactivity *(… when Moore meets Einstein…)*
  - Many faults in communication systems are transient
- **Permanent faults** remain in the system until they are repaired; e.g., a broken wire or a software design error.
- **Intermittent faults** are transient faults that occur from time to time
  - E.g. a hardware component that is heat sensitive, it works for a time, stops working, cools down and then starts to work again

Failure Modes



Approaches to Achieving Reliable Systems

- **Fault prevention** attempts to eliminate any possibility of faults creeping into a system before it goes operational
- **Fault tolerance** enables a system to continue functioning even in the presence of faults

- Both approaches attempt to produces systems which have well-defined failure modes

Fault Prevention

- Two stages: fault avoidance and fault removal
- Fault avoidance attempts to limit the introduction of faults during system construction by:
  - use of the most reliable components within the given cost and performance constraints
  - use of thoroughly-refined techniques for interconnection of components and assembly of subsystems
  - packaging the hardware to screen out expected forms of interference.
  - rigorous, if not formal, specification of requirements
  - use of proven design methodologies
  - use of languages with facilities for data abstraction and modularity
  - use of software engineering environments to help manipulate software components and thereby manage complexity

Fault Removal

- In spite of fault avoidance, design errors in both hardware and software components will exist
- Fault removal: procedures for finding and removing the causes of errors; e.g. design reviews, program verification, code inspections and system testing
- System testing can never be exhaustive and remove all potential faults
  - **Remember: a test can only be used to show the presence of faults, not their absence!**
  - It is sometimes impossible to test under realistic conditions
  - Most tests are done with the system in simulation mode and it is difficult to guarantee that the simulation is accurate
  - Errors that have been introduced at the requirements stage of the system's development may not manifest themselves until the system goes operational

Failure of Fault Prevention Approach

- In spite of all the testing and verification techniques, hardware components will fail; the fault prevention approach will therefore be unsuccessful when
  - either the frequency or duration of repair times are unacceptable, or
  - the system is inaccessible for maintenance and repair activities

- An extreme example of the latter is the Mars Pathfinder and the Mars Rovers Spirit and Opportunity, http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

- Alternative is **Fault Tolerance**

Levels of Fault Tolerance

- **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 of fault tolerance required will depend on the application
- Most safety critical systems require full fault tolerance, however in practice many settle for graceful degradation

Redundancy

- All fault-tolerant techniques rely on extra elements introduced into the system to detect & recover from faults
- Components are redundant as they are not required in a perfect system
- Often called protective redundancy
- Aim: minimise redundancy while maximising reliability, subject to the cost and size constraints of the system
- Warning: the added components inevitably increase the complexity of the overall system
- This itself can lead to less reliable systems
- E.g., first launch of the space shuttle
- It is advisable to separate out the fault-tolerant components from the rest of the system

SOME SOFTWARE ISSUES

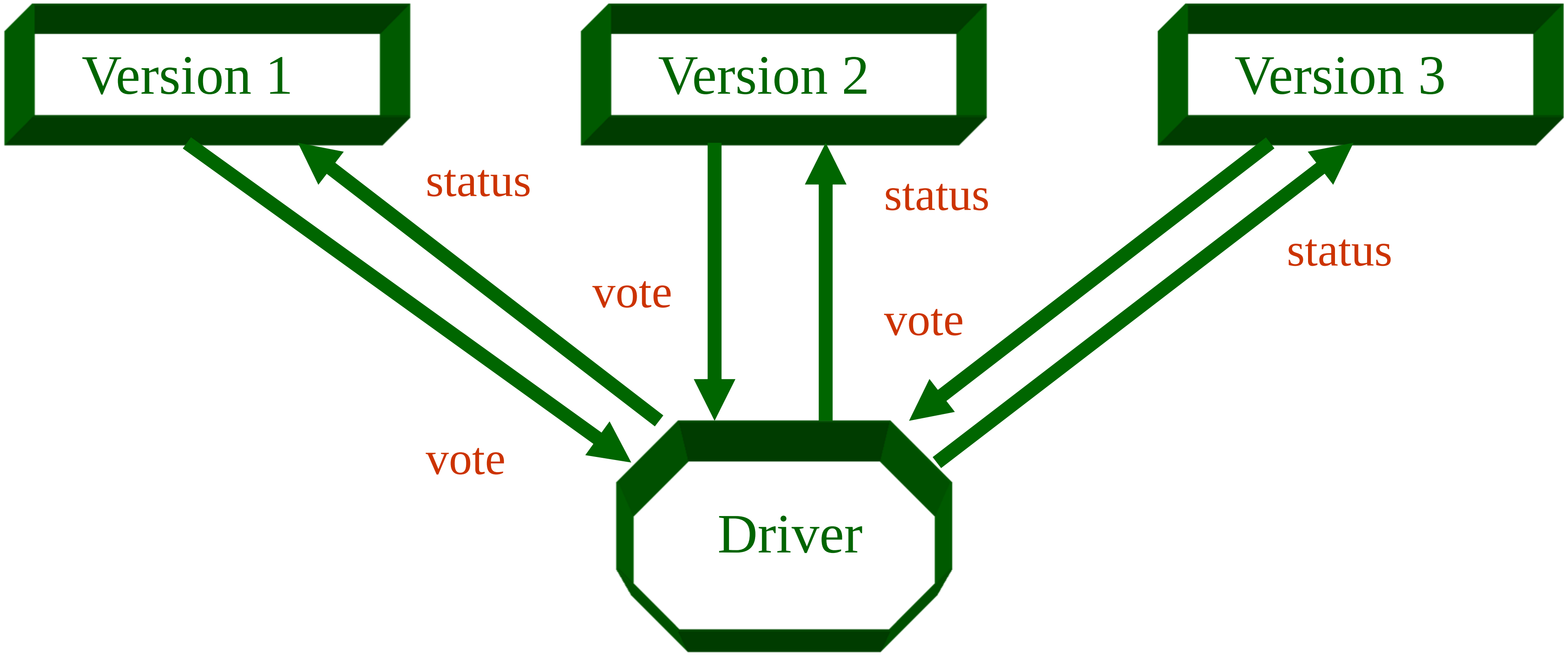
Software Fault Tolerance

- Used for detecting design errors
- Static — N-Version programming
- Dynamic
  - Detection and Recovery
  - Recovery blocks: backward error recovery
  - Exceptions: forward error recovery

N-Version Programming

- Design diversity
- The independent generation of N (N > 2) functionally equivalent programs from the same initial specification
- No interactions between groups
- The programs execute concurrently with the same inputs and their results are compared by a driver process
- The results (VOTES) should be identical, if different the consensus result, assuming there is one, is taken to be correct

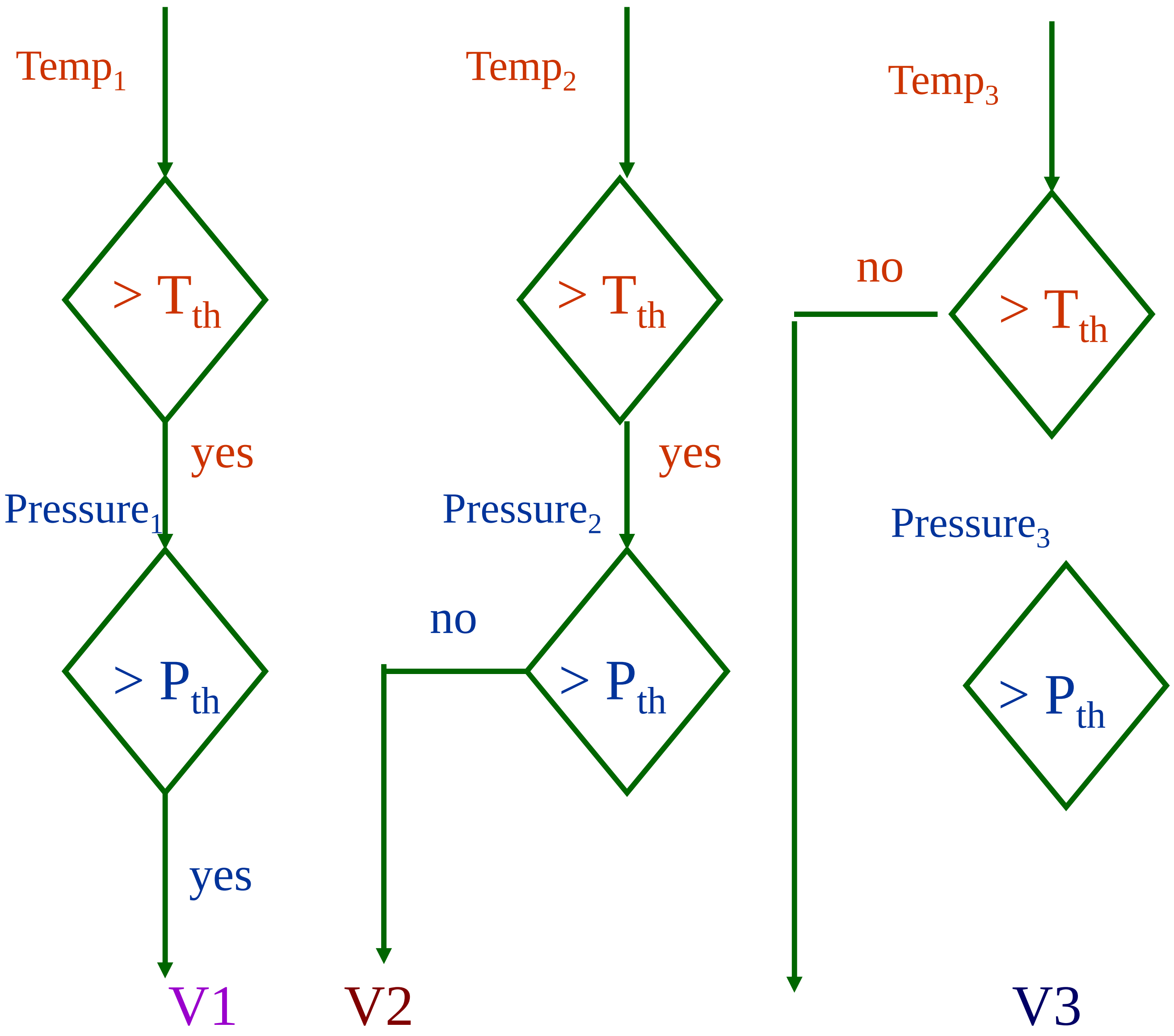
N-Version Programming



Vote Comparison

- To what extent can votes be compared?
- Text or integer arithmetic will produce identical results
- Real numbers => different values
- Need inexact voting techniques

Consistent Comparison Problem



This example illustrates 3-version V1, V2, V3 (triplicate) software for a process control system which monitors temperature and pressure sensors and then takes appropriate actions according to their values to ensure the integrity of the system. The 3 systems vote on the outcome. As a result of finite-precision arithmetic, each version will calculate slightly different values. The consistent comparison problem occurs when both readings are around their threshold values.

N-version programming depends on:

- Initial specification — The majority of software faults stem from inadequate specification? A specification error will manifest itself in all N versions of the implementation
- Independence of effort — Experiments produce conflicting results. Where part of a specification is complex, this leads to a lack of understanding of the requirements. If these requirements also refer to rarely occurring input data, common design errors may not be caught during system testing
- Adequate budget — The predominant cost is software. A 3-version system will (at least) triple the budget requirement and cause problems of maintenance. Would a more reliable system be produced if the resources potentially available for constructing an N-versions were instead used to produce a single version?

Software Dynamic Redundancy

Four phases:

- error detection — no fault tolerance scheme can be utilised until the associated error is detected
- damage confinement and assessment — to what extent has the system been corrupted? The delay between a fault occurring and the detection of the error means erroneous information could have spread throughout the system
- error recovery — techniques should aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality)
- fault treatment and continued service — an error is a symptom of a fault; although damage repaired, the fault may still exist

Error Detection

- Environmental detection
  - hardware — e.g. illegal instruction
  - O.S/RTS — null pointer
- Application detection
  - Replication checks
  - Timing checks
  - Reversal checks
  - Coding checks
  - Reasonableness checks
  - Structural checks
  - Dynamic reasonableness check

Damage Confinement and Assessment

- Damage assessment is closely related to damage confinement techniques used
- Damage confinement is concerned with structuring the system so as to minimise the damage caused by a faulty component (also known as firewalling)
- Modular decomposition provides static damage confinement; allows data to flow through well-define pathways
- Atomic actions provides dynamic damage confinement; they are used to move the system from one consistent state to another

Error Recovery

- Probably the most important phase of any fault-tolerance technique
- Two approaches: forward and backward
- Forward error recovery continues from an erroneous state by making selective corrections to the system state
- This includes making safe the controlled environment which may be hazardous or damaged because of the failure
- It is system specific and depends on accurate predictions of the location and cause of errors (i.e, damage assessment)
- Examples: redundant pointers in data structures and the use of self-correcting codes such as Hamming Codes

Backward Error Recovery (BER)

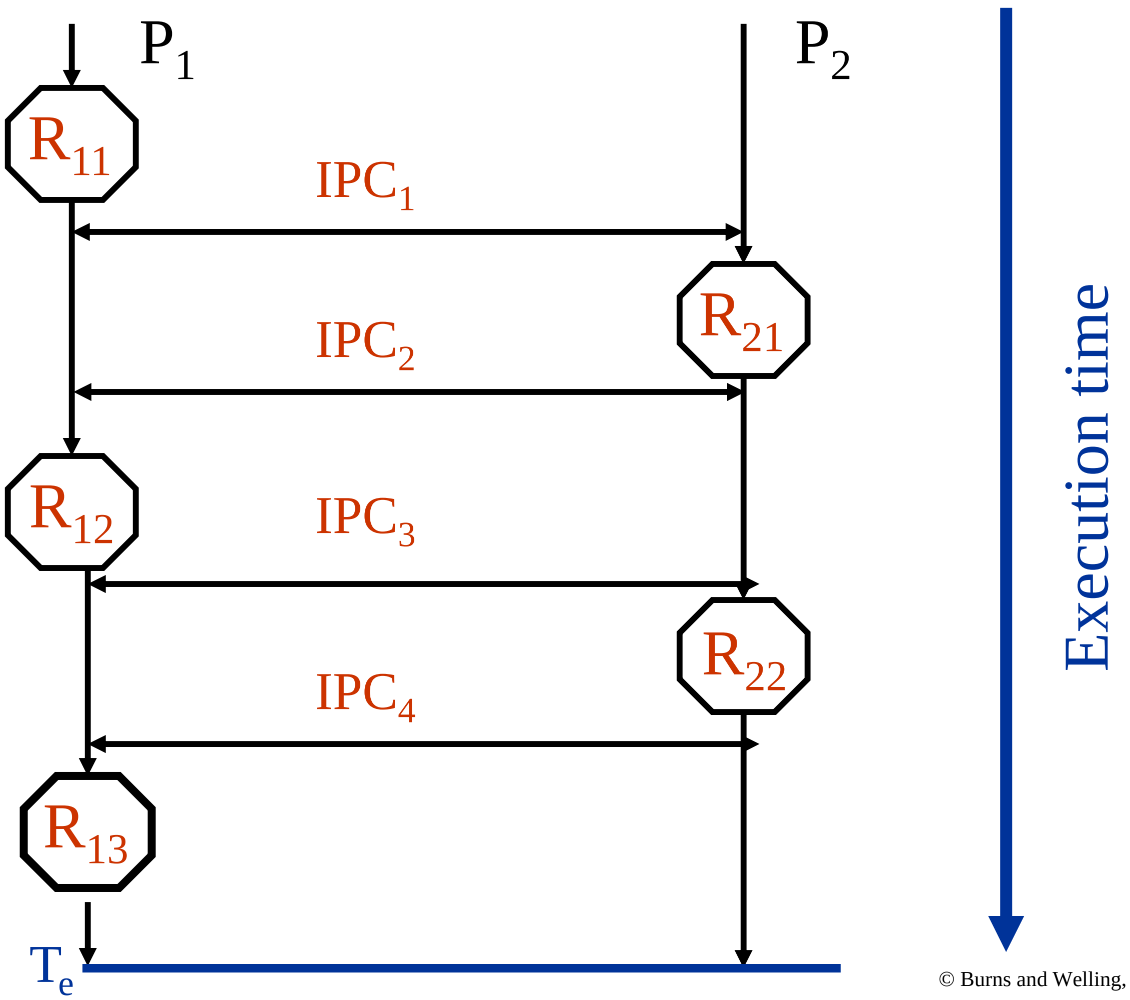
- BER relies on restoring the system to a previous safe state and executing an alternative section of the program
- This has the same functionality but uses a different algorithm (c.f. N-Version Programming) and therefore no fault
- The point to which a process is restored is called a recovery point and the act of establishing it is termed checkpointing (saving appropriate system state)
- Advantage: the erroneous state is cleared and it does not rely on finding the location or cause of the fault
- BER can, therefore, be used to recover from unanticipated faults including design errors
- Disadvantage: it cannot undo errors in the environment!

The Domino Effect

- With concurrent processes that interact with each other, BackwardErrorRecovery is more complex. Consider:

If P_1 detects an error at T_e, then simply roll back to recovery point R_{13}.

However, consider the case where P_2 detects an error at T_e. If P_2 is rolled back to R_{22}, then it must undo the communication IPC_4 with P1, which requires P_1 to roll back to R_{12}. But then P_2 must be rolled back to R_{21}, etc etc.



Fault Treatment and Continued Service

- Error Recovery returned the system to an error-free state; however, the error may recur; the final phase of Fault Tolerance is to eradicate the fault from the system
- 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 2 stages: fault location and system repair
- Error detection techniques can help to trace the fault to a component. For, hardware the component can be replaced
- A software fault 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!

The Recovery Block approach to Fault Tolerance

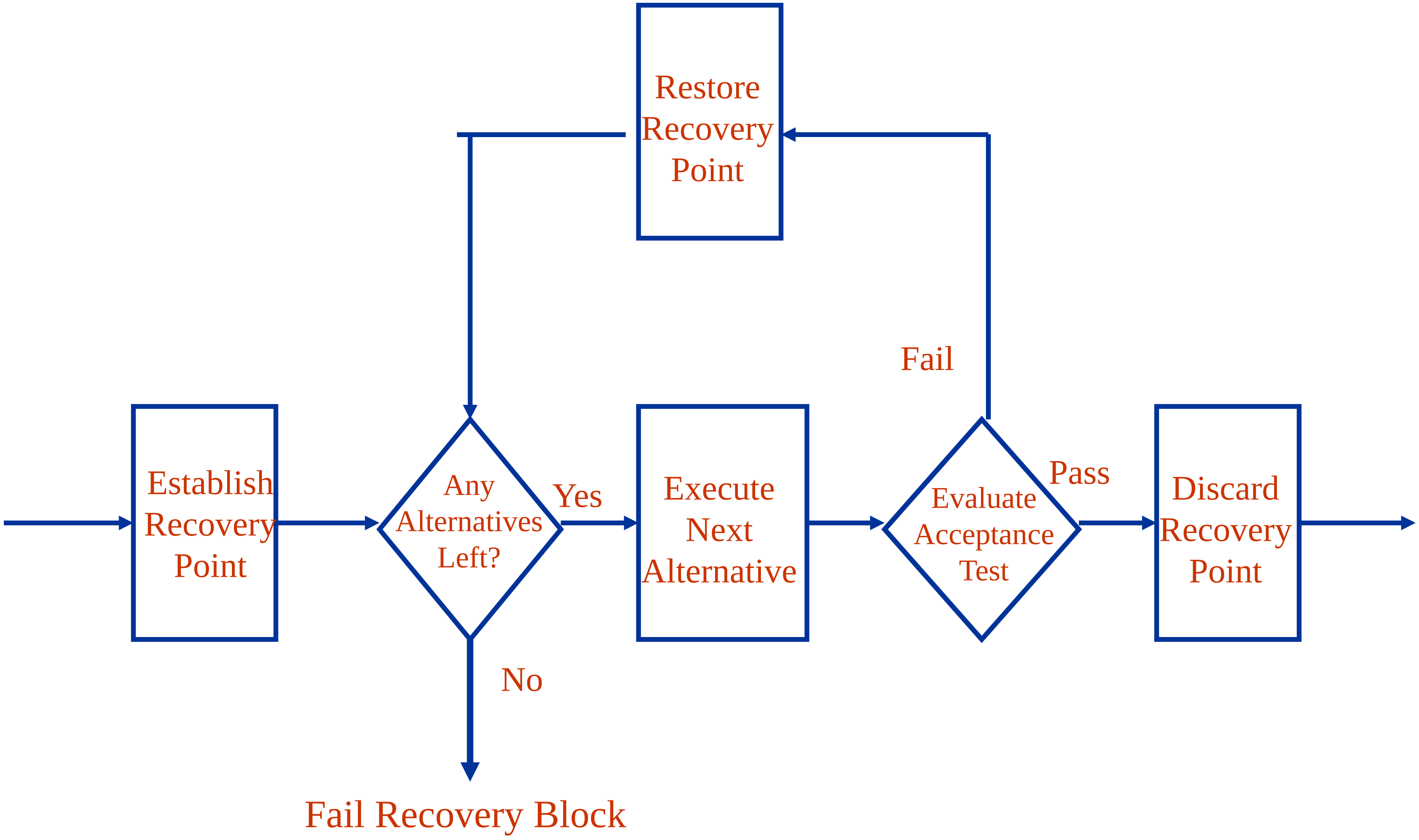
- Recovery blocks: language support for BER
  - The Recovery block concepts was introduced some 40 years ago. For more info search the web
- At the entrance to a block is an automatic recovery point and at the exit an acceptance test
- The acceptance test is used to test that the system is in an acceptable 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 Syntax

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

- Recovery blocks can be nested
- If all alternatives in a nested recovery block fail the acceptance test, the outer level recovery point will be restored and an alternative module to that block executed

Recovery Block Mechanism



The Acceptance Test

- The acceptance test provides the error detection mechanism which enables the redundancy in the system to be exploited
- The design of the acceptance test is crucial to the efficacy of the RB scheme
- There is a trade-off between providing comprehensive acceptance tests and keeping overhead to a minimum, so that fault-free execution is not affected
- Note that the term used is acceptance not correctness; this allows a component to provide a degraded service
- All the previously discussed error detection techniques discussed can be used to form the acceptance tests
- However, care must be taken as a faulty acceptance test may lead to residual errors going undetected

N-Version Programming vs Recovery Blocks

- Static (NV) versus dynamic redundancy (RB)
- Design overheads — both require alternative algorithms, NV requires driver, RB requires acceptance test
- Runtime overheads — NV requires N * resources, RB requires establishing recovery points
- Diversity of design — both susceptible to errors in requirements
- Error detection — vote comparison (NV) versus acceptance test(RB)
- Atomicity — NV vote before it outputs to the environment, RB must be structure to only output following the passing of an acceptance test

Dynamic Redundancy and Exceptions

- An exception can be defined as the occurrence of an error
- Bringing an exception to the attention of the invoker of the operation which caused the exception, is called raising (or signally or throwing) the exception
- The invoker's response is called handling (or catching) the exception
- Exception handling is a forward error recovery mechanism, as there is no roll back to a previous state; instead control is passed to the handler so that recovery procedures can be initiated
- However, the exception handling facility can be used to provide backward error recovery

Exceptions (incl. signals)

Exception handling can be used to:

- cope with abnormal conditions arising in the environment
- enable program design faults to be tolerated
- provide a general-purpose error-detection and recovery facility

BUS ERROR

- Bus errors are usually signaled with the SIGBUS signal, but SIGBUS can also be caused by any general device fault that the computer detects. A bus error rarely means that the computer hardware is physically broken—it is normally caused by a bug in a source code. There are two main causes of bus errors:
  – **non-existent address**. The CPU is instructed by software to read or write a specific physical memory address. Accordingly, the CPU sets this physical address on its address bus and requests all other hardware connected to the CPU to respond with the results, if they answer for this specific address. If no other hardware responds, the CPU raises an exception, stating that the requested physical address is unrecognized by the whole computer system. Note that this only covers ***physical*** memory addresses. Trying to access an undefined ***virtual*** memory address is generally considered to be a segmentation fault rather than a bus error, though if the MMU is separate, the processor can't tell the difference.

BUS ERROR (cont)

- unaligned access
  - Most CPUs are *byte-addressable*, where each unique memory address refers to an 8-bit byte. Most CPUs can access individual bytes from each memory address, but they generally cannot access larger units (16 bits, 32 bits, 64 bits and so on) without these units being "aligned" to a specific boundary. For example, if multi-byte accesses must be 16 bit-aligned, addresses 0, 2, 4, and so on would be considered aligned and therefore accessible, while addresses 1, 3, 5, and so on would be considered unaligned. Similarly, if multi-byte accesses must be 32-bit aligned, addresses 0, 4, 8, 12, and so on would be considered aligned and therefore accessible, and all addresses in between would be considered unaligned. Attempting to access a unit larger than a byte at an unaligned address can cause a bus error.

unaligned memory access C code

from Wikipedia (not VxWorks compliant!)

```
int main(int argc, char **argv)
{
        int *iptr;
        char *cptr;

#if defined(__GNUC__)
#  if defined(__i386__)
        /* Enable Alignment Checking on x86 */
         __asm__("pushf\norl $0x40000,(%esp)\npopf");
# elif defined(__x86_64__)
        /* Enable Alignment Checking on x86_64 */
        __asm__("pushf\norl $0x40000,(%rsp)\npopf");
# endif
#endif

/* malloc() always provides aligned memory */
        cptr = malloc(sizeof(int) + 1);

/* Increment the pointer by one, making it misaligned */
        iptr = (int *) ++cptr;

 /* Dereference it as an int pointer, causing an unaligned access */
        *iptr = 42;

        return 0;
 }
```

SEGMENTATION/PAGE FAULT

- A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a read-only location, or to overwrite part of the operating system).
- Common causes:
  - On Unix-like operating systems, a signal called SIGSEGV is sent to a process that accesses an invalid memory address. On Microsoft Windows, a process that accesses invalid memory receives the STATUS_ACCESS_VIOLATION exception.
    - attempting to execute a program that does not compile correctly. Note that most compilers will not output a binary given a compile-time error.
    - a buffer overflow.
    - using uninitialized pointers.
    - dereferencing NULL pointers.
    - attempting to access memory the program does not own.
    - attempting to alter memory the program does not own (storage violation).
    - exceeding the allowable stack size (possibly due to runaway recursion or an infinite loop)

Summary

- Reliability: a measure of the success with which the system conforms to some authoritative specification of its behaviour
- When the behaviour of a system deviates from that which is specified for it, this is called a failure
- Failures result from faults
- Faults can be accidentally or intentionally introduced into a system
- They can be transient, permanent or intermittent
- Fault prevention consists of fault avoidance and fault removal
- Fault tolerance involves the introduction of redundant components into a system so that faults can be detected and tolerated

Summary

- N-version programming: the independent generation of N (where N >= 2) functionally equivalent programs from the same initial specification
- Based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently
- Dynamic redundancy: error detection, damage confinement and assessment, error recovery, and fault treatment and continued service
- Atomic actions to aid damage confinement

Summary

- With backward error recovery, it is necessary for communicating processes to reach consistent recovery points to avoid the domino effect
- For sequential systems, the recovery block is an appropriate language concept for BER
- Although forward error recovery is system specific, exception handling has been identified as an appropriate framework for its implementation
- The concept of an ideal fault tolerant component was introduced which used exceptions
- The notions of software safety and dependability have been introduced

”The five nines” (99.999% uptime)