CS_313 High Integrity Systems/ CS_M13 Critical Systems

Course Notes
Additional Material
Chapter 6: Fault Tolerance

Anton Setzer
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/∼csetzer/lectures/critsys/11/index.html

December 8, 2011

No Additional Material

For this subsection no additional material has been added yet.

(b) Types of Faults

- Faults can be characterised by the following criteria:
  - Nature (random/systematic).
  - Duration.
  - Extent.

Classification by Nature

Faults can be classified by their nature: random vs. systematic faults:

- **(i) Random faults.**
  - Random faults are faults in components of a system, which occur with a certain probability.
  - We can predict random faults by collecting statistical data from large numbers of samples of similar components.
  - Random faults are usually hardware faults.
    - Reason for random faults are that any hardware is subject to environmental influences, which might affect its correct operation.
    - E.g. radioactivity, radiation, humidity, warmth, cold, wear and tear.

Classification by Nature

- In software reliability engineering one considers as well software errors as random.
- Because random faults can be predicted well, they are more easy to tolerate.

Classification by Nature

- **(ii) Systematic Faults**
  - Systematic faults are faults which are not random.
    - Either a component has it, or it doesn't have it.
  - Software faults are usually considered to be systematic.
  - Three kinds of systematic faults:
    - Mistakes in the specification of a system.
    - Mistakes in the implementation of software.
    - Mistakes in the design of hardware.
  - Difficult to tolerate.
    - E.g. if two programmers write the same program, it might be that both make the same systematic mistake.

Classification by Duration

Faults can be classified by their duration:

- **(i) Permanent faults**.
  - Remain in existence indefinitely, until corrective action is taken.
  - Software faults are always permanent.
  - Many hardware component faults are permanent.

Classification by Duration

- **(ii) Transient faults**.
  - Appear, and vanish again.
  - Typical example are effects of radioactive particles hitting a semi conductor of a memory chip.
    - If it happens, the state of a few bits is changed.
    - But there is no lasting damage to the chip.
  - Although infrequent and not lasting, one needs to take steps to correct this error before a system error is caused.

Classification by Duration

- **(iii) Intermittent faults**.
  - Appear, disappear, and then reappear after some time.
  - Results of
    - poor solder joints, corrosion on connector contacts.
      At some times connections are possible, at others not.
    - electromagnetic radiation.
    - **Electromagnetic compatibility (EMC)** is the ability of a system to work correctly in the presence of (electromagnetic) interference from other electrical equipment, and not to interfere with other equipment or other parts itself.

Classification by Duration

- Problems of electromagnetic radiation occur
  - within wires of a digital circuits
  - between computers and other sources of noise (usually caused by direct electromagnetic radiation or by coupling through common power lines).
  - Particular problems close to car engines, jet engines, nuclear power reactors, high-power electric motors.
  - Mobile phones and CD/DVD players cause nowadays problems (e.g. not allowed in planes).
- **Software errors**, especially those caused by race conditions, often appear to be intermittent, but are **always permanent**.

Classification by Extent

Faults can be classified by their extent:

- **(i) Localised faults** affect only a single hardware or software module.
- **(ii) Global faults** have effects that permeate through the entire system.

(c) Fault Models

- In order to analyse the effect of **hardware faults** on the entire system, one uses **fault models**.
- These are **not perfect representations** of what is physically actually happening.
- However, they help to design **test procedures**, to simulate **fault conditions**, and to develop **fault tolerant** systems.
  - Testing of safety critical software needs to take into account that safety requirements are met even if one or two hardware components fail.

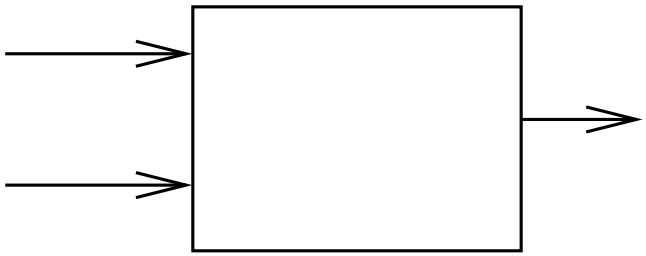
Fault Models

- We consider 3 fault models:
  - **Single-stuck-at fault model**.
  - **Bridging fault model**.
  - **Stuck-open fault model**.

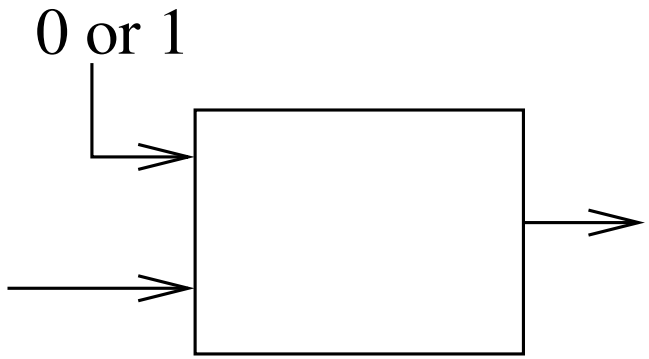
Single-Stuck-At Model

- **Single-stuck-at model** assumes that a fault within a module causes it
  - to respond as if one of its inputs or outputs is stuck at logic 0 or 1.
  - and such that the basic functionality of the circuit is otherwise unaffected.

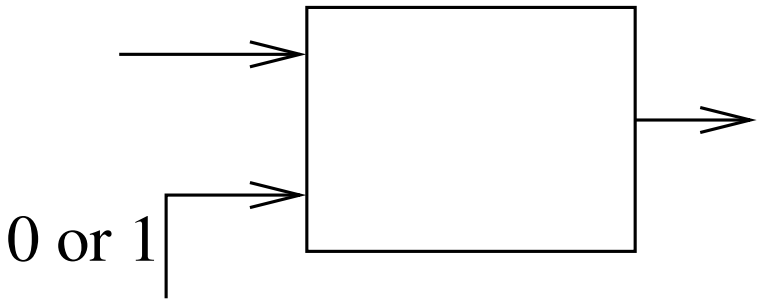
Example (Single-Stuck-At Model)



Module uneffected



Input 1 stuck at 0 or 1

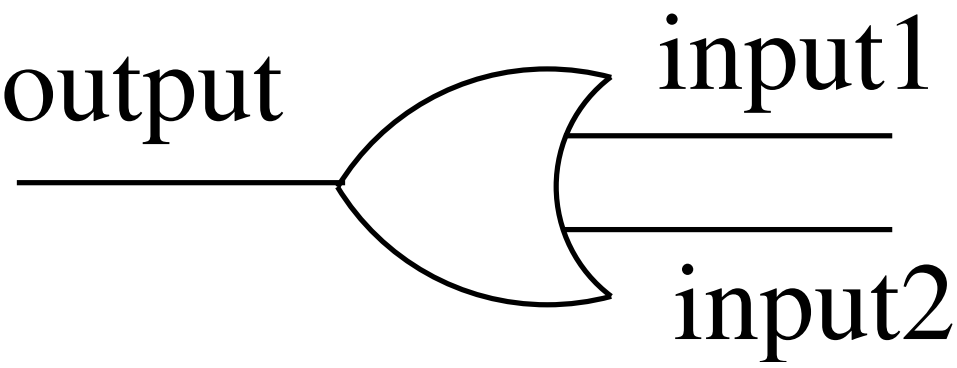


Input 2 stuck at 0 or 1

Output stuck at 0 or 1

Example: Or-Gate

- Assume for instance an or gate with inputs input1, input2 and output output:



- If input1 is stuck at 0 then the output is computed as follows:

$$\text{output} = 0 \vee \text{input2} = \text{input2}$$

Example: Or-Gate

- If input1 is stuck at 1, then the output is constant 1:

$$\text{output} = 1 \vee \text{input2} = 1$$

  - This is identical to the situation where the output is stuck at 1.

Analysis of Single-Stuck-At Model

- Majority of faults arising from broken tracks, open or short-circuit components and shorts between tracks can be represented by the single-stuck-at model.
- Single-stuck-at model cannot represent accurately transient or intermittent faults.
- **Complexity of testing**:
  - For a circuit with N nodes there are only 2N single-stuck-at faults:
    one for each node stuck at 0,
    one for each node stuck at 1.
  - Exhaustive testing feasible.

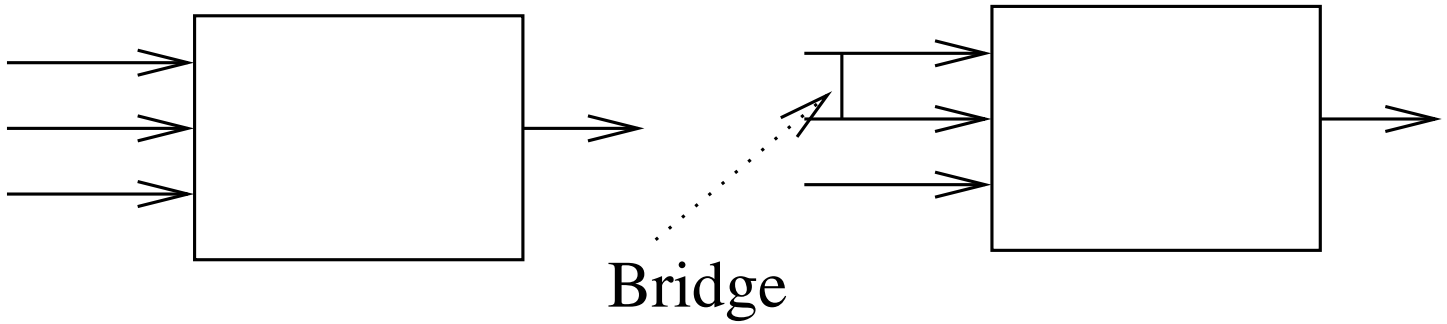
Bridging Model

- A **bridging** or **short-circuit fault** occurs when two or more nodes in a circuit are accidentally joined together to form a permanent fault.
- In positive logic, i.e.
  - 1 represented by power on,
  - 0 represented by power off,

  a bridging fault between two inputs has the effect of both inputs being ANDed together (see next slide).
- In negative logic, i.e.
  - 1 represented by power off,
  - 0 represented by power on,

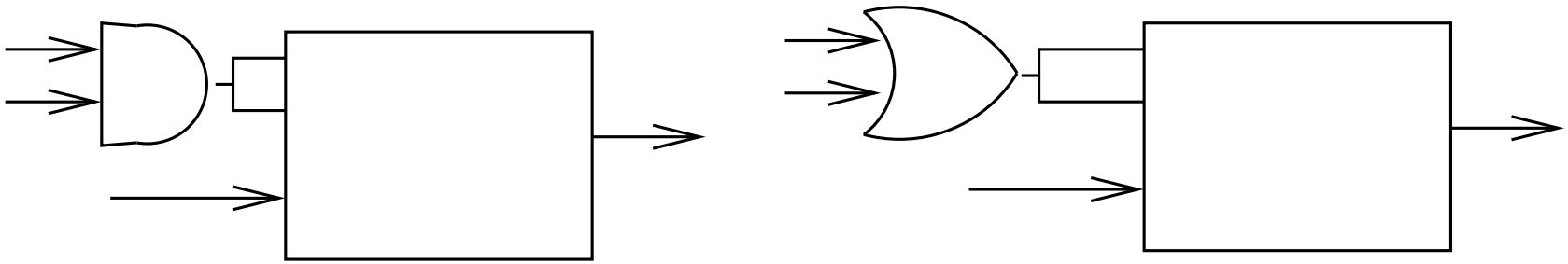
  a bridging fault between two inputs has the effect of both inputs being ORed together (see next slide).

Simple Example (Bridging Fault)



Module unaffected

Bridging fault



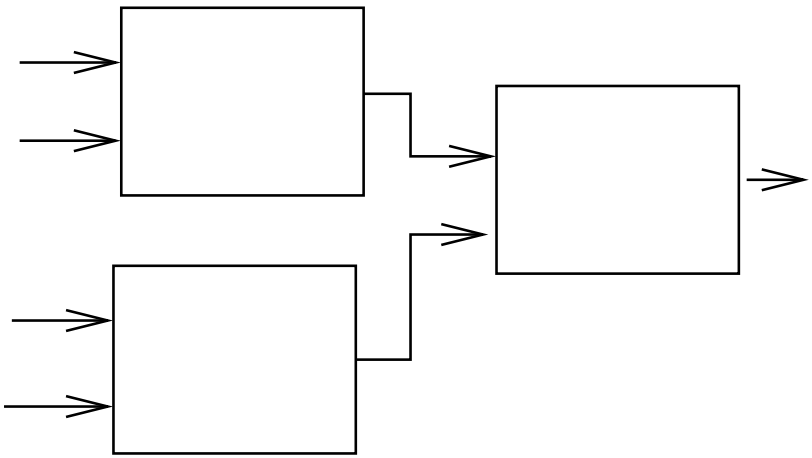
Effect in positive Logic

Effect in negative Logic

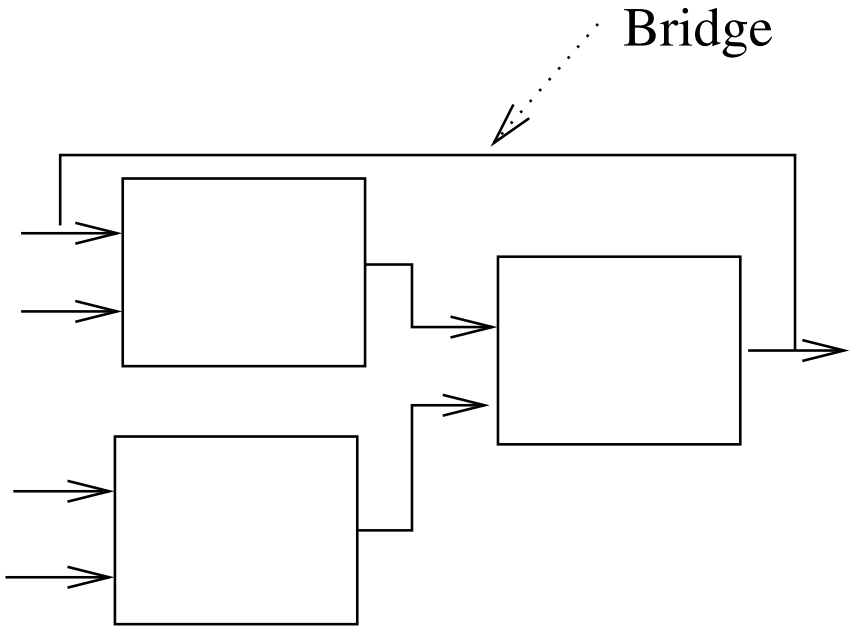
More Complex Bridging Faults

- Bridges between inputs and outputs might result in converting combinatorial circuits into sequential ones, and might result in instability or oscillation.

Complex Example (Bridging Fault)

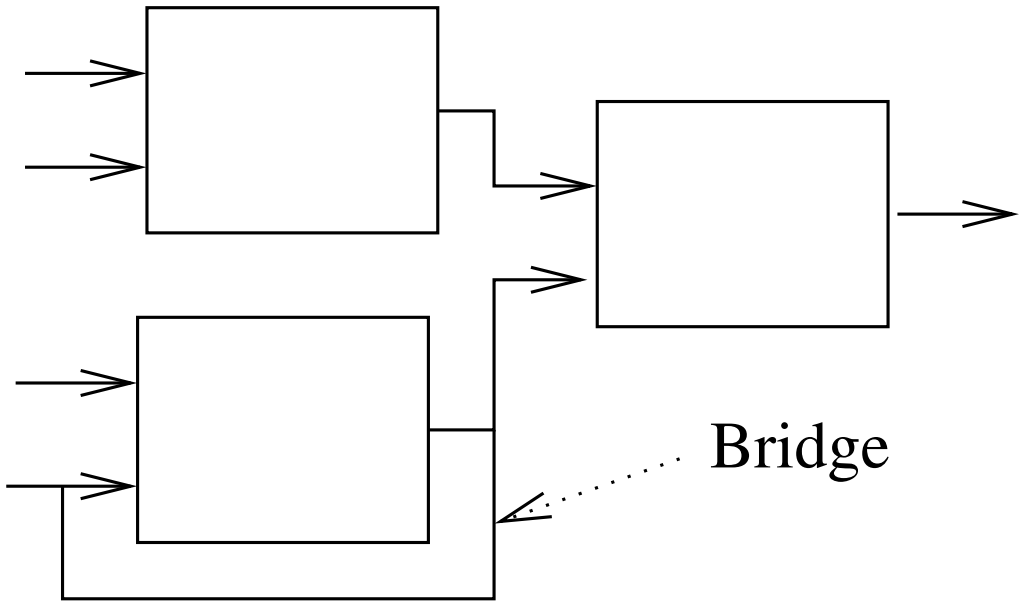


Unaffected



Bridging fault 1

Complex Example (Bridging Fault)



Bridging fault 2

Analysis of Bridging Model

- Bridging faults behave usually different from single-stuck-at faults.
- **Complexity of testing** more complex:
  - For a circuit with N nodes there are $\binom{N}{M}$ bridging faults between M notes at the same time.
    - Especially, there are $\binom{N}{2} = \frac{N \cdot (N-1)}{2}$ bridging faults between two nodes.
    - Makes exhaustive testing impossible in most cases.

The Stuck-Open Model

- **Stuck-open** fault occurs if in a CMOS gate, both output transistors are turned off because of an internal open- or short-circuit.
- Therefore output is neither pulled to high nor to low.
- Depending on the exact fault, the gate will alternate between
  - driving the output
  - or maintaining its previous output.
- Length of time it maintains its output depends on the gate and the nature of the fault.
- Therefore circuit gets a complex sequential characteristic.

Use of Fault Models

- Exhaustive testing of circuits is not feasible except for simple combinatorial circuits.
- Using fault models, test vectors can be developed which test for faults occurring by one of the above fault models.
  - Testing only feasible by assuming single failures.
  - E.g. a circuit with N nodes can have $3^N - 1$ multiple stuck-at faults, which is infeasible to test.
    Why $3^N - 1$ faults?
    - Each of the nodes can be error free, stuck at 1 and stuck at 0, giving 3^N possibilities.
    - The only case when the circuit is correct is when all nodes are error free. Excluding it we get $3^N - 1$ faulty cases.

Use of Fault Models

- Testing for bridging faults usually infeasible as well.
  Usually restriction to testing for single occurrences of single-stuck-at faults.

- Fault models can be used for developing strategies for tolerating faults.
- **Limitations:**
  - Hardware design faults (especially wrong logic design) are usually not be covered by those fault models.
  - Software faults are usually not covered by these models.
    - In software reliability engineering, fault models for software are developed.

(d) Fault Coverage

- **Fault coverage** is the fraction of possible faults that can be avoided, removed, detected or tolerated.
  - Usually it is difficult to give a numerical estimate, except when good fault models can be used.
- **Fault removal coverage** is the fraction of faults found during the testing phase of system development.
  - Testing vectors aim at 100% fault removal coverage for the faults in the underlying fault model.
  - However, fault models never include all possible faults.
  - Especially, most models cover only single faults and don't cover transient or intermittent faults.
  - Therefore, fault removal coverage is never 100%.

Fault Coverage

- **Fault detection coverage** is the ability of a system to detect faults during operation.
  - Using fault models, fault detection coverage can be estimated, but we have the same limitations as above.
- **Fault tolerance coverage** is the ability of a system to tolerate faults.

No Additional Material

For this subsection no additional material has been added yet.

(f) Fault Detection Techniques

- Many fault-tolerance techniques rely on detection of faults.
- In practice often difficult to detect faults – one can often only detect errors created by faults.
- Sometimes it is not even necessary to detect the exact faults – detecting that an error occurred as a consequence, suffices.
  - E.g. in case of transient faults, detection of faults is not necessary. One only needs to know that a fault has occurred, in order to deal with the consequences.
- We consider some software and hardware techniques for fault detection.

Functionality Checking

- **Functionality checking** is the use of software or hardware routines in order to check whether the hardware of a system is functioning correctly.

Examples of Functionality Checking

- Checking of RAM done by writing into memory and reading it back again, and checking whether the value stored is reproduced.
  - **Problem 1**: It is in general not feasible to check the complete memory space.
  - **Problem 2**: Stuck-at faults might be not detected, if the checking test vector checks for the value at which memory is stuck-at (so it always returns that value).
    - Can be overcome by checking with two test vectors, of which the second is the negation of the first one.

Examples of Functionality Checking

- **Problem 3**: Even if a memory location is non-existent, it might be that for a short time due to the capacitance of the bus the test vector is still reproduced.
  - Can be overcome by interleaving writes and reads to different locations.
- Other problems, which can be overcome by using sophisticated test routines.

Examples of Functionality Checking

- Checking of processors done by executing sequences of calculations and comparing them with known results (usually stored in ROM).
- Checking of connections in multiprocessor systems by checking that each processor can communicate with its neighbours.

Misc. Checking Methodologies

- **Consistency checking** uses knowledge about the nature of the information within the system, e.g. that data must be within a certain range (**range checking**).
- **Signal comparison** checks in systems with redundancies the signals at various points in the modules and compares them.
  - **Checking pairs** is a special case of signal comparison. Here one checks whether the outputs of identical modules with the same inputs are identical.
- **Information redundancy** uses error-detecting codes in order to detect errors in the data given.

Instruction Monitoring

- **Instruction monitoring** is checking for illegal instructions.
  - If the binary code is corrupted, one might (especially if the opcode is corrupted) obtain an illegal instruction.
  - Some processors immediately raise an exception, others might proceed.
    - Some processors use illegal instructions for internal testing purposes (this is often not documented), and therefore will not raise an exception.
    - For critical systems one should use processors which raise an exception in such cases.

Loopback Testing

- **Loopback testing** means that one sends a signal arriving at its destination back to its origin and checks whether the original signal and the signal sent back coincide.
  - Finds stuck-at failures in the signal lines.
  - If the outward and return path are too close, then the outward path might influence the return path and transmit its content even so the path is broken.

Watchdog Timers

- **Watchdog timers** are used in order to check whether the processor has crashed.
  - The watchdog timer starts with a certain value and decrements periodically.
  - If the watchdog timer has reached zero, the processor will be reset.
  - The processor regularly resets the watchdog timer, before it reaches 0.
  - If the processor crashes, it will in most cases (but not all) not reset the watchdog timer, and therefore the processor will be reset.

Watchdog Timers

- **Limitations:**
  - It takes a few milliseconds before a crash is detected.
    - During that period the results of the processor are wrong and this might cause hazards.
  - It might be that the processor crashes in such a way that the watchdog timer is reset periodically.

Bus Monitoring

- In **bus monitoring** one checks that the addresses send to the bus are in a certain range.
- Allows to detect, if the processor has crashed and is executing illegal instructions.

Power Supply Monitoring

- **Power supply monitoring** is not directly a fault detection mechanism but a fault prevention mechanism.
  - One checks whether the power is in a certain range, especially above a certain limit (against to high voltage usually overvoltage protection is sufficient).
  - If it is out of range, the processor is instructed to take measurements in order to protect its data (and in order not to produce incorrect output).
- In some cases one needs an **uninterruptible power source** using large-capacity batteries which are all the time reloaded in order to provide power in the event of a power supply failure.

No Additional Material

For this subsection no additional material has been added yet.

(h) Software Fault Tolerance

- In this subsection we will consider how to tolerate faults by using software.
- We will consider two techniques:
  - (i) **N-version programming.**
  - (ii) **Recovery blocks.**

(i) N-Version Programming

- N-version programming means that N different versions of the software are written.
  - All fulfil the same specification.
    - Note that this doesn't help if there is a problem in the specification.
  - Written by different teams or companies, and maybe using different tools, languages or techniques.
- The N versions are run on the same input data
  - on the same processor (interleaved)
  - or on separate processors (operating in parallel).
- The results are compared using a software implementation of one of the techniques introduced in the section on hardware redundancy.

N-Version Programming

- **Problem** of N-version programming:
  - Additional development costs.
  - Additional processing power needed for running several versions of the program and for the voting process.
- N-version programming only used for very critical applications, e.g.
  - Airbus 330/340,
  - Space shuttle.

(ii) Recovery Blocks

- **Recovery block technique** based on **acceptance tests**.
  - Acceptance tests are software versions of fault detection.
  - Acceptance test check the consistency of the output of one software module (a unit inside the program like a procedure, package or a class).

Examples of Acceptance Tests

- Check whether output is within certain boundaries.
- Check for run time errors (e.g. arithmetic over- or underflow),
- Check for excessive execution time.
- Check for arithmetic correctness by reversing the computation.
  - E.g., if a module computes the square root, the acceptance test checks whether the square of the output is equal to the input.

Recovery Blocks (Cont.)

- For critical modules, several implementations are developed, of which one is the main one.
- Then the following is executed:
  - Execute the main module.
  - Carry out acceptance test.
  - If this fails, switch to an alternative module.
  - Carry out acceptance test.
  - If this fails, switch to the second alternative module.
  - Etc.
  - If everything fails, raise an error.
- Recovery blocks can be considered as a software version of the standby spare arrangement.

Recovery Blocks

- Problem: if one module fails, one has to make sure that any side-effects carried out by it are reversed.
- Done by establishing a **recovery point** at the beginning of the module, and by introducing a mechanism in order to make sure that one can switch back to the recovery point in case of failure of the acceptance test.
- Storing the state of all variables when reaching the recovery point too expensive and time consuming in general.
- Instead one stores only those variables which are possibly changed by the module.

Recovery Blocks

- It was suggested to provide a special language construct as follows:

```
ensure     <acceptance test>
by         <main module>
else by    <alternative module 1>
else by    <alternative module 2>
                 ...
else by    <alternative module n>
else       error
```

No Additional Material

For this subsection no additional material has been added yet.

No Additional Material

For this subsection no additional material has been added yet.