Safety-Critical Computer Systems

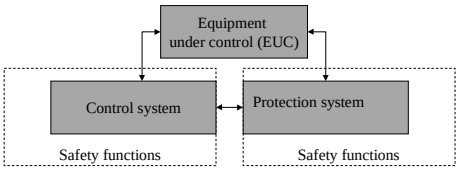
Treatment of System Faults

Simin Nadjm-Tehrani
www.ida.liu.se/~snt

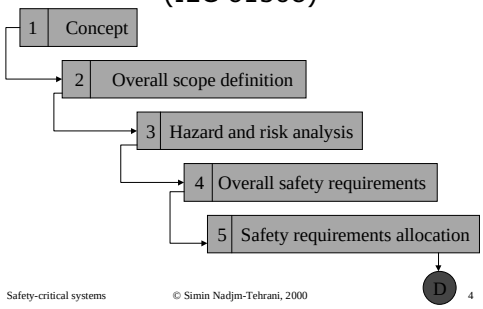
First some news...

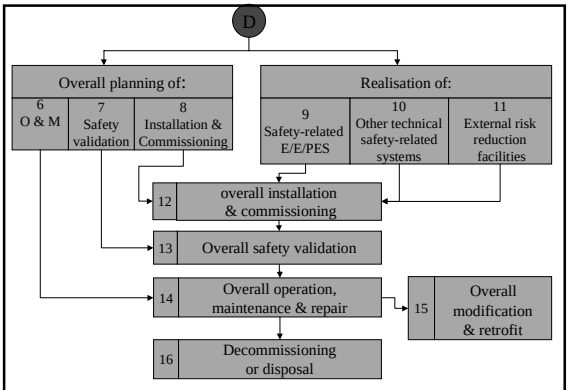
- Groups formed and group meetings planned
- Fault-tree+ installed, can be used for testing and learning
- Last lecture/resource session: discussions with Volvo

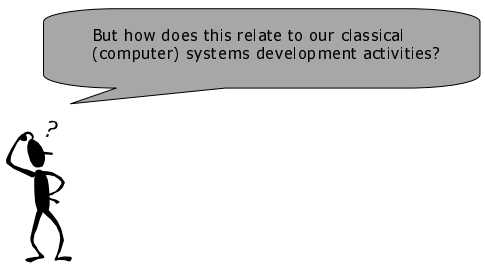
Structure of safety-critical systems



Overall safety lifecycle (IEC 61508)







Types of faults

- Transient
  - e.g. electromagnetic radiation in environment
- Intermittent
  - e.g. loose wire
- Permanent
  - e.g. design error, hardware defect

Verification and validation

- Show that system behaviour is in accordance to requirements specification
  - Inspections
  - Testing
  - Formal verification
- Show that requirements are complete and consistent

On-line treatment of faults (fault-tolerance)

- Fault detection
  - by program or environment
- Fault containment with the help of redundance in
  - software
  - hardware
  - data

From article in Edinburgh Review, 1824: D. Lardner

"The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if their computations are carried out by different methods."*

* *people who compute*

System architecture

- Highly influences choice of methods for incorporation of redundancy
- Monolithic/Distributed systems
- Synchronous/Asynchronous systems

Static Redundancy

To be used all the time (whether errors showed up or not), just in case...

- SW: N-version programming
- HW: Voting and masking
- Data: Parity bits, checksums

N-version programming

- Main problem: to get the different versions to act differently at test instances which are error-inducing
- Night/Leveson experiment:
  - the only errors missed, were missed by all 28 versions!

Dynamic Redundancy

Used when error has occured and must be contained

- SW: recovery methods
- HW: switch to back-up modules
- Data: self-correcting codes

Error recovery

Backward:

- roll back the system to a safe state which was reached before the error appeared (when did error appear?)
- restart with alternative module (how is the result affected by earlier module´s side effects?)

Error recovery

Forward:

- ”fix the error” and continue as if nothing happened
- redundacy lies where one fixes the error

Distributed systems

- Introduce new complications
  - no global clock
  - richer fault models
  - network partitions
- Software replication and group mechanisms
  - transparency in treatment of faults

Exception management

- Every program should test for validity range of its computations, why exceptions?
- Mechanism to support recovery models in programming languages
  - e.g. Ada´s exceptions support backward error recovery via their termination model

Static and dynamic exceptions

- Is there a difference between foreseen and unforeseen faults?
- For foreseeable faults, recover via specific exception handlers, clean up operations
- For unforeseen faults, achieve graceful degradation


```
package Temp_Control is
subtype Temperature is integer range 0..100;
     Sensor_Dead, Actuator_Dead: exception;
...
end Temp_Control;
package body Temp_Control is

     procedure Set_Temp(...) is
     begin
          -- set new value for actuator
          if No_Response then
               raise Actuator_Dead
          end if;
     end Set_Temp;
...
end Temp_Control;
```



```
     function Read_Temp return Temperature is
     begin
          -- read sensor value
          if No_Response then
               raise Sensor_Dead
          end if;
          -- return the value
          exception
               when Constraint_Error =>
               -- too high a temperature
               -- take appropriate action
     end Read_Temp;
...
end Temp_Control;
```



```
begin
     -- initialize
     Set_Temp(...);
     when  Actuator_Dead =>
          -- take some action
     ...
end Temp_Control;
```