Safety Critical Software Engineering

- As software is used more widely, safety has become an important issue.
- Safety aspects of software are poorly understood at the moment.
- Software itself is poorly understood still.
- Safety analysis can not be usually undertaken piecemeal – a whole system approach is normally required.
- Safety analysis is context sensitive:
  - a system “safe” in one context may not be in another
    - eg. a computer controlled ventilation system for use in glasshouses might not be suitable for use in a multi-story office block....
  - Why?
Case Study 1 – F16A

- “The world’s most sophisticated, foolproof and deadly combat aircraft yet.”
- Computer monitoring of all critical operations.
  - Computer would ignore pilot directives if they would result in operation outside pre-set envelope.
- Failure 1: gear-up while landed
  - Solution: software now checks “squat switches”
- Failure 2: gear-down while cruising (gear down at >300 knots is fatal)
  - Solution: software now checks airspeed
- Failure 3: gear won’t come up
  - Solution: correct race condition introduced by failure 2 fix which latched boolean flags for squat switches.
- Morals:
  - The piecemeal approach is clearly ineffective,
  - “Quick Fixes” often aren’t...

Case Study 2 – the Space Shuttle

- “The world’s fastest flying brick”
- Key design aspect: fail-operational
  - If a failure occurs, the system can continue
- Original design: 4 computers in control with a fifth “overseeing”
  - Fail-operational/fail-operational/fail-safe design
- All five computers are independently coded, black and white box tested
- The four computers “vote” hydraulically!
  - They all operate hydraulic actuators independently
  - If one pushes the wrong way, the others can overcome the force and “vote” it out.
  - An erroneous computer is shutdown.
- All operations are prioritized, if there are not enough compute resources available, low priority tasks are shed.
Shuttle – Why it doesn’t Work

• Uses slow, robust 1970’s technology
  - Large transistor size to minimise α particle damage
• The PASS system is essentially a distributed real time control system
  - Four computers each see everything
  - Independently decide on action
  - Come to a consensus view about what to do.
• Question: How many computers do you need to have to guarantee consensus with at most k of them failed?
• Answer: more than you think
  - 3k + 1
• The shuttle design is not fail-operational, perhaps not even fail safe.
• Moral: Concurrency, replication and fault tolerance are really tough.

Case Study 3 – Patriot Missile

• Overview:
  - One missile battery failed to lock onto a scud missile which subsequently killed personnel at a Saudi military camp in Dhahran
• Cause:
  - Time interval calculations were imprecise, due to the 24 bit registers in the hardware.
• Why did the programmers miss this?
  - It doesn’t matter till the system has been up continuously for many hours, 30 mins of “testing” didn’t detect this.
• Moral: You need to consider the hardware limitations too. You need to be aware of the implicit assumptions you are making (yes, we can figure the time by differencing the two register values...
Case Study 4 – Therac 25

- A radio-therapy machine which has killed several patients by massive overdose.
- The Therac-25 is a classic example of a poorly engineered software system.
- Implemented on a PDP-11 system, written in assembly language.
- Many of the problems result from race conditions due to the use of shared structures with no synchronisation.
- Moral: lack of whole system testing and methodical testing overall.

Case Study 5 – Satellite Control

- Not really safety critical
- Satellite control systems are very cryptic
  - Geostationary satellites perform station keeping operations once a week or so
  - A computer tells the operator how many Newtons of force and for what duration and direction are required.
  - The operator then types this information into the groundstation computer
  - When data entry is done, the operator verifies the data and sends the information to the satellite.
- Many cases of incorrect transcription have been documented, resulting in reduced satellite service life.
- Moral: very poor interface design encourages human error.
- It is exactly this kind of transcription error which crashed the NZ DC-10 into Mt. Erebus.
Case Study 6 – Series 3000 Railcars

- The new STA railcars are electric, with diesel powered on-board power generation
- Anti-lock brakes
  - Uses regenerative braking until wheel lock
  - Switches to air-brakes and controls wheel lock
- Brakes take 2 seconds to switch
  - Control system came from the SNCF underground trains
  - 2 seconds was OK there
- Fundamental problem: emergency brakes also operate through the ABS system
  - Drivers refused to drive the trains - they were unsafe - documented cases of > 4Km stopping distance at 60Km/h
  - Rail conditions are different here from France!
- Moral: Testing in context is required. The implicit assumptions the designers made were no longer true.

Case Study 7 – Airbus A300

- Birmingham airport had an A300 hit the terminal (remember the airport movies?)
- The primary cause was a “safety” feature:
  - If you are at a “rotation” point, the aircraft must be under full power
- A plane was being tugged across a taxiway
  - The pilots had “told” the computers the plane was about to take off
  - The pilot applied the brakes
  - The tug pushed
  - The front nose wheel “unweighted”....
  - The safety system detected a rotation and wound the engines to full power.....it also notified the pilot that the airspeed instrumentation was faulty and sounded an alarm....
- Not an “unknown” problem - An FA-18A had previously taken off while being serviced in a hanger from Edwards Airforce base....
Safety

• Clearly there are many examples where safety has been compromised – even by very experienced development teams/
• Why?
  - Safety is difficult to ensure.
  - You have predict all possible risks.
  - It isn’t just factors within the control of the software that may need to be dealt with.
  - It is expensive, and many clients are ant prepared to pay for the costs of managing safety.
  • Ethical dilemma – what would you do if you were asked to develop safety critical software within a budget that could not assure the safety of the end users?

Specification

• Safety and security concerns often manifest themselves as requirements that are difficult to classify as functional or non-functional.
• Often best described as a “shall not” requirement.
  - The system shall not allow users to modify access permissions on any files that they have not created (security).
  - The system shall not allow the simultaneous activation of more than three alarm signals (safety).
• Often these shall not requirements are decomposed into specific functional requirements.
• Formal methods are likely to be cost effective for safety-critical systems.
Reliability

- Reliability is a complex concept which should be considered at the system level rather than the individual component level.
  - Components are interdependent.
  - Failure in one component can be propagated to another.
- Three dimensions must be considered when specifying overall system reliability:
  - Hardware reliability - what is the probability of hardware failure and how long does it take to repair the component.
  - Software reliability - how likely is it that a software component will produce incorrect output?
    - Software failure differs from hardware failure because software does not wear out, and the software can operate correctly even after an incorrect result is produced.
  - Operator reliability - How likely is it that the operator of the system will make an error?

Reliability

- These three are closely linked. A hardware failure, could cause the software to generate an incorrect result. Unexpected system behavior could cause the operator stress. Operator error is most likely when the operator is under stress.
- Simplistically, if a system depends on component A and component B with failure probabilities $P_A$ and $P_B$, then the overall probability of system failure $P_S$ is:
  $$P_S = P_A + P_B$$  (additive)
- As the number of system components increases, the overall probability of system failure increases.
- To increase reliability, a number of critical components may be replicated. The components work together and the component group is operational as long as at least one of the components works correctly.
- Simplistically, if the probability of a component failing is $P_A$ and all components are independent, and the component is replicated $n$ times then the chance of system failure is
  $$P_S = P_a^n$$  (multiplicative)
Error avoidance

- In addition to recognizing risks during specification, we need to be aware of common errors during implementation that could potentially cause failure. Should we ban the use of these constructs on this project?
- Apart from goto’s there are a number of well documented constructs and programming techniques that are inherently error prone.

1. Floating-point numbers: imprecision and rounding errors.
4. Parallelism: timing interactions between processes.
5. Recursion: logic errors and use of all system resources.
6. Interrupts: interrupt may cause a critical operation to be terminated.
7. Inheritance: all the code associated with an object is not in one place resulting in difficulty in understanding and programming errors. Combined with dynamic binding, it can lead to run-time timing problems.
8. Aliasing: multiple names for the same entity. A change to one variable may inadvertently change the value associated with another.
9. Default input processing: security loophole as an attacker could present program with unexpected inputs that are not rejected by the program.
Error avoidance

- Some programming languages are designed to reduce the risk from some of these features.
  - Java does not employ pointers, it does not have goto statements, it does its own garbage collection.
- Some standards (company policy) prohibit the use of these constructs. These are often not practical.
- To develop software with minimal faults, it is essential to have a software development process that is well defined, repeatable and that includes a spectrum of verification and validation activities.
  - A well defined process is one that has been standardized and documented.
  - A repeatable process is one that does not rely on individual interpretation and judgment.

Fault minimization

- The process should include a well planned, comprehensive testing process as well as other activities whose aim is fault detection.
- Validation activities geared toward fault minimization include:
  - Requirements inspection
    - Intended to discover faults in the system specification.
    - A high proportion of faults in delivered software result from requirements errors.
  - Requirements management
    - Track changes in requirements.
    - Trace these through the design and implementation.
    - Many errors in delivered systems result from a failure to ensure that a requirements change has actually been included in the design and implementation of the system.
Fault minimization

- Model checking
  - Automatic analysis of system models with CASE tools that check for internal and external consistency.
  - Internal consistency means that a single model is (self) consistent.
  - External consistency means that different models of the system are consistent (e.g., object model, state model).

- Design and code inspections
  - Often based on checklists of common faults.
  - Aim to identify and rectify these faults before system testing.

Fault minimization

- Static analysis
  - Automated technique of program analysis where a program is analyzed in detail to find potentially erroneous conditions.

- Test planning and management
  - A comprehensive set of tests for the system should be designed and the testing process itself should be carefully managed to ensure complete test coverage and traceability between the system tests and the system requirements and design.
Process assurance

• Quality assurance in the system development process is particularly important for safety-critical systems.
  1. Accidents are rare events in critical systems and it may be practically impossible to simulate them during the testing of the system.
  2. Safety requirements are sometimes “shall not” requirements that exclude unsafe behavior. It is impossible to demonstrate conclusively through testing and other validation activities that these requirements have been met.

Process assurance

• Explicit attention must be paid to safety during all stages of the software process. Safety assurance activities must be included in the process:
  1. Hazard logging and monitoring system must be in place to trace hazards from preliminary hazard analysis through testing and system validation.
  2. Appoint project safety engineers who have explicit responsibility for all safety aspects of the system.
  3. Extensive use of safety reviews.
  4. Creation of a safety certification system whereby safety-critical components are formally certified for their assessed safety.
  5. Use of a detailed configuration management system to track all safety related documentation and keep it in step with all the technical documentation.