



System and Software Safety

Systems Certification and Integrity
Directorate of Aviation Engineering
Directorate General Technical Airworthiness

1

Directorate General Technical Airworthiness (DGTA-ADF)



Overview

- Introduction to System Safety
 - Relationship to Design Assurance
- Software Safety Requirements
 - Software Safety Requirements by example.
 - Explore a Rudimentary Flight Control System
 - Safety Critical Functions
 - Generic Software Safety Requirements
 - Software Hazard Analysis and Software Safety Requirements
 - V&V and Software Safety Requirements
 - Safety Case and Software Safety Requirements
 - A Real World Case Study

2

Directorate General Technical Airworthiness (DGTA-ADF)





Idealist Approach

“To specify and design a ‘perfect’ system, which cannot go wrong because there are no faults in it, and to prove that there are no faults in it.”

Pragmatic Approach

“To aim for the first philosophy, but to accept that mistakes may have been made, and to include error detection and recovery capabilities to prevent errors from actually causing a hazard to safety.”



System and Software Safety Aims

- **System Safety Engineering**
 - ... identify and eliminate hazards in order to reduce the associated risk. [MIL-STD-882C]
- **Software Safety Engineering**
 - ... software will execute within the system context with an acceptable level of safety. [JSSSCSSH]





Overview of System Safety



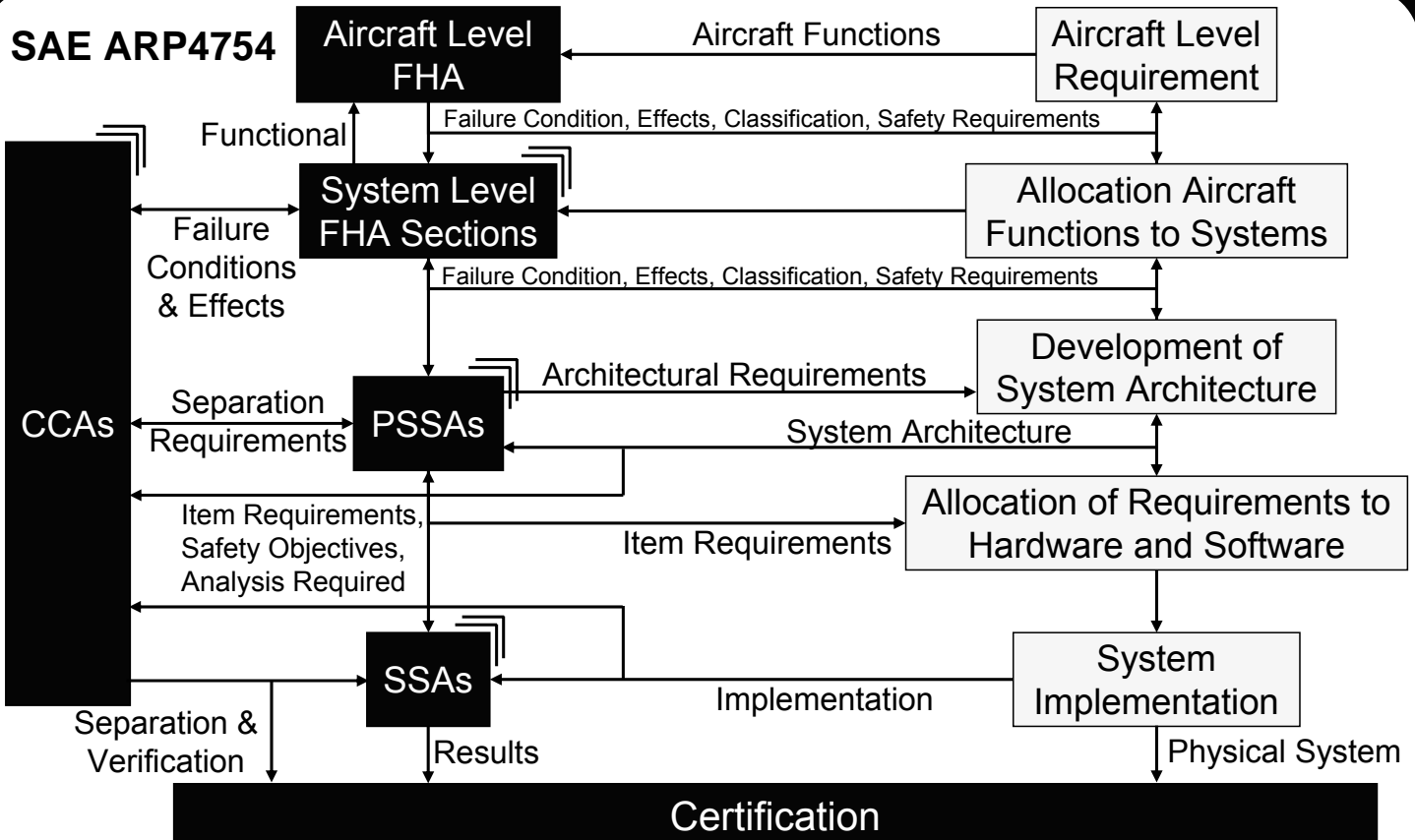
System Safety Process

- **Identify Hazards**
 - Top Down Approach
 - What does the system do?
 - How can it fail?
 - Bottom Up Approach
 - How can components fail?
 - What is the effect of a component failure?
- **Evaluate Hazards**
 - Qualitative Methods
 - Combination of failure mode severity and probability
 - Quantitative Methods
 - e.g. 10^{-9} per flight hour for Catastrophic failures
 - Risk Acceptance Matrix
- **Treat Hazards**
 - System Safety Order of Precedence
 - Design for Minimum Risk
 - Incorporate Safety Devices
 - Provide Warning Devices
 - Develop Procedures and Training
 - Other Options
 - Retain Risk (OAA) – Issue Papers
 - Avoid Risk

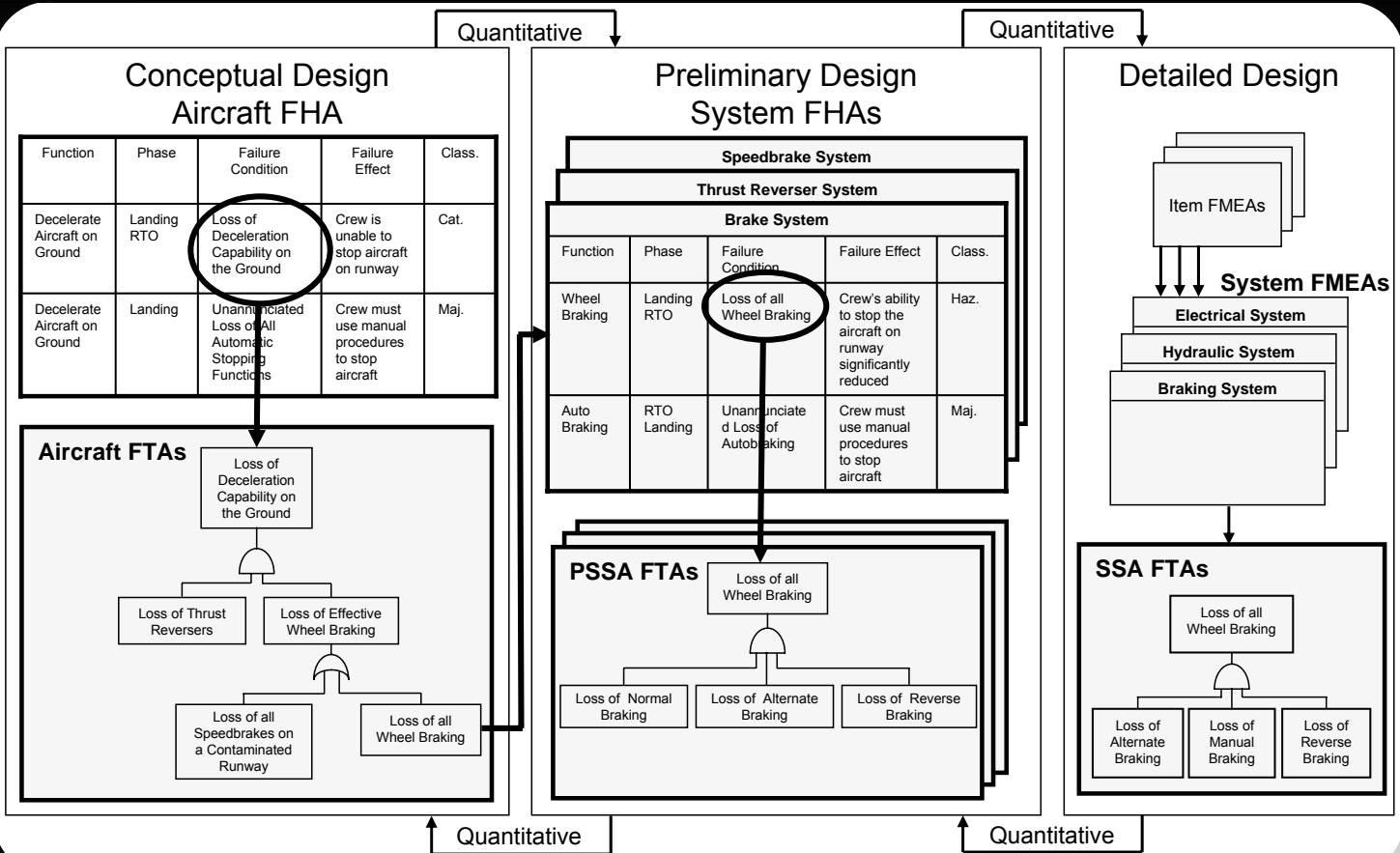




System Safety Assessment Process Model



Top Down Bottom Up Approach





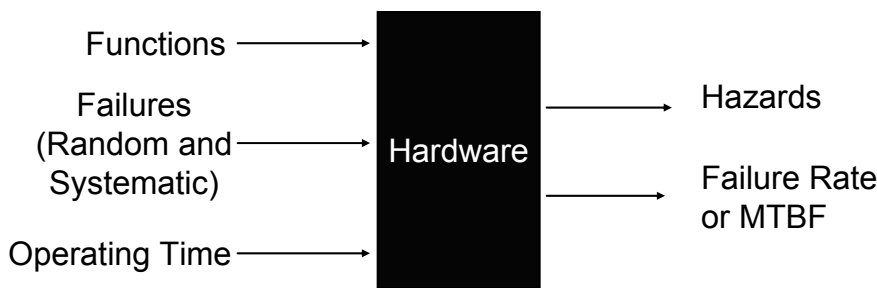
Hazard Risk Index Matrix

Probability	Severity			
	Catastrophic	Hazardous	Major	Minor
Infrequent	1	2	4	7
Remote	3	5	8	11
Extremely Remote	6	9	12	14
Extremely Improbable	10	13	15	16

1-3	Unacceptable
4-6	Requires mitigation, unless acceptance given by the DAR and OAAR
7-10	Acceptable, with approval from the designated Commonwealth authority
11-16	Acceptable with approval from the designated Contractor authority

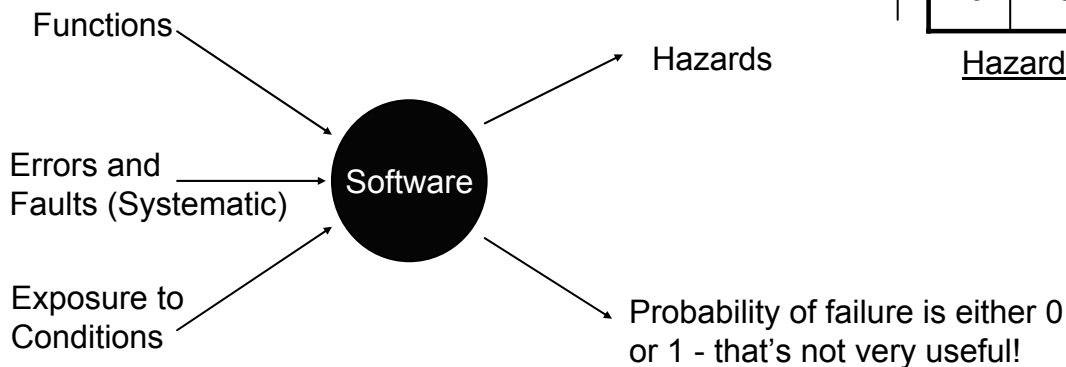


Hazard Risk Index for Software



Probability	Severity				
	15	10	6	3	1
19	14	9	5	2	
22	18	13	8	4	
24	21	17	12	7	
25	23	20	16	11	

Hazard Risk Index Table

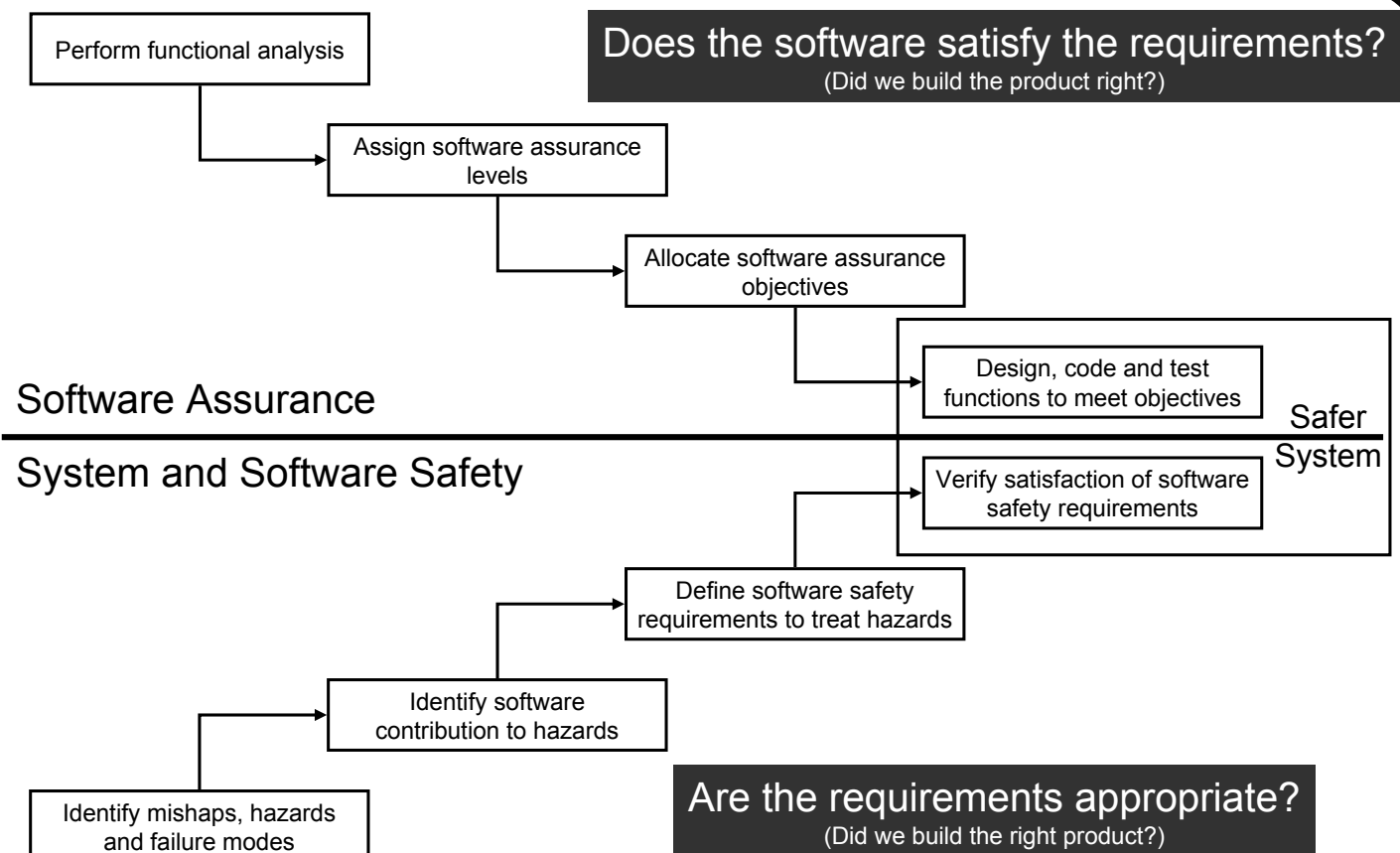




Software Safety



Complementary Approaches





Software Safety Requirements

- **Safety-related software faults arise most often from:**
 - Discrepancies between documented requirements specifications and the requirements needed for correct functioning of the system.
 - Misunderstandings about the software's interface with the rest of the system.
- **Software-related accidents have occurred when the software satisfied its specification and when the operational reliability of the software was very high.**
 - Requirements specify behaviour that is not safe from a system perspective.
 - Requirements do not specify some particular safety behaviour.
 - The software has unintended (and unsafe) behaviour beyond what is specified in the requirements.
- **i.e.: The software satisfies the requirements, but the requirements were wrong or incomplete.**



Software Safety Requirements

- **Therefore:**
 - The identification and allocation of software safety requirements in the context of the system are key to the realisation of acceptably safe software intensive systems.
- **That's all very well, but how does one go about it?**
 - Is there a formal approach?
 - How do we understand software in the system context?
 - I know how to write software that meets the specification, doesn't that mean it's safe?
 - What other requirements do I need?





What is software safety?

- Software safety is the analysis of a software driven system that is conducted in order to identify safety requirements.
 - The software shall ... (do something safe).
 - The software shall not ... (do something dangerous).
- Software safety is *not* the assignment of software assurance levels.
- How does software safety differ from system safety?
 - The focus: software safety considers each software item, system safety considers an entire system.
 - Otherwise, no real difference. Similar techniques, similar goals.



Identifying Software Safety Requirements

- Analyse the system and software in order to identify:
 - requirements that detect and handle erroneous inputs to the software system
 - e.g. sanity checks on input data
 - requirements that detect and handle software faults or erroneous outputs from the software system
 - e.g. output voting schemes
 - requirements to detect and handle common software failures
 - e.g. watchdog timer on processor activity





An Example

- We'll look at a rudimentary flight control system.
- Process:
 - Identify system functions.
 - Identify system level functional failure modes and associated severities.
 - Identify potential software failure modes.
 - Define software safety requirements to detect and handle or prevent software failure modes.
 - Define generic software safety requirements.
 - Verify that software satisfies allocated requirements.



Identify Flight Control System Functions and Failure Modes





How to identify failure modes?

- Consider:
 - ‘Loss of’
 - What happens if the system stops working and I detect that it has stopped working?
 - ‘Malfunction without warning’
 - What happens if the system continues to work, but does the wrong thing and I can’t detect it?



Failure Conditions – ‘Loss of’

- Control of Pitch, Roll and Yaw
 - Catastrophic. Hazardous for loss of one lateral axis.
- Control Lift and Drag
 - Control Flap/Slat – Major or Hazardous
 - Spoiler Control – Major or Hazardous
 - Gear Position – Major
- Automatic Control of Flight
 - Minor (with warning) Major (without warning)
- Stability Augmentation
 - Variable – depends on flight characteristics of aircraft





Failure Conditions – ‘Malfunction without Warning’

- Control of Pitch, Roll and Yaw
 - Catastrophic.
- Control Lift and Drag
 - Control Flap/Slat – Hazardous or Catastrophic
 - Spoiler Control – Hazardous or Catastrophic
 - Gear Position – Major
- Automatic Control of Flight
 - Major (single axis limited authority), Hazardous (multi-axis limited authority), Catastrophic (unlimited authority).
- Stability Augmentation
 - Variable – depends on flight characteristics of aircraft



Likely Safety Critical Software Functions

- Any function which controls or directly influences the pre-arming, arming, enabling, release, launch, firing, or detonation of a weapon system, including target identification, selection and designation.
- **Any function that determines, controls, or directly influences the flight path of a weapon system.**
- Any function that controls or directly influences the movement of gun mounts, launchers, and other equipment, especially with respect to the pointing and firing safety of that equipment.
- Any function which controls or directly influences the movement of munitions and/or hazardous materials.
- Any function which monitors the state of the system for purposes of ensuring its safety.
- Any function that senses hazards and/or displays information concerning the protection of the system.
- Any function that controls or regulates energy sources in the system.

Source: JSSSC SSSH





Likely Safety Critical Software Functions

- **Fault detection priority.** The priority structure of fault detection and restoration of safety or correcting logic shall be considered safety-critical. Software units or modules handling or responding to these faults.
- **Interrupt processing software.** Interrupt processing software, interrupt priority schemes and routines that disable or enable interrupts.
- **Autonomous control.** Software components that have autonomous control over safety critical hardware.
- **Software controlled movement.** Software that generates signals which have been shown through analysis to directly influence or control the movement of potentially hazardous hardware components or initiate safety-critical actions.
- **Safety-critical displays.** Software that generates outputs that displays the status of safety critical hardware systems. Where possible, these outputs shall be duplicated by non-software generated output.
- **Critical data computation.** Software used to compute safety-critical data. This includes applications software that may not be connected to or directly control a safety-critical hardware system (e.g., stress analysis programs).

Source: JSSSC SSSH



Identify Software Failure Modes and Software Safety Requirements



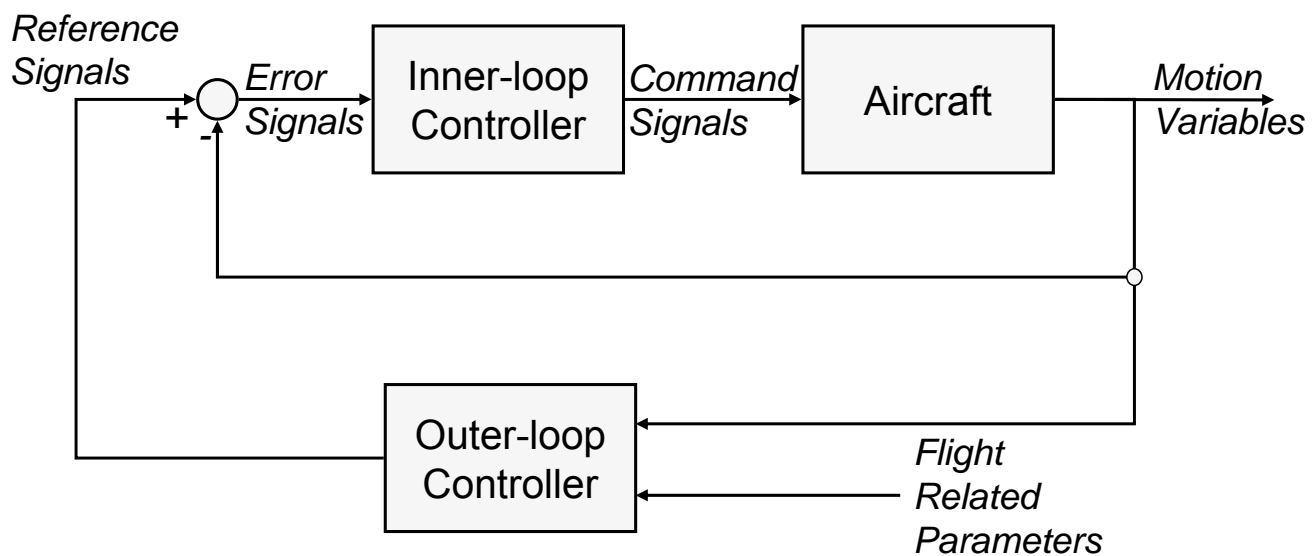


Aims of Software Hazard Analysis

- Identify software contribution to system level hazards.
- Identify software failure modes that are hazardous in the system context.
 - May identify hazards that weren't identified at the system level.
- Define software safety requirements to prevent, or detect and handle, software failure modes.
- Produce evidence to support the system safety case.

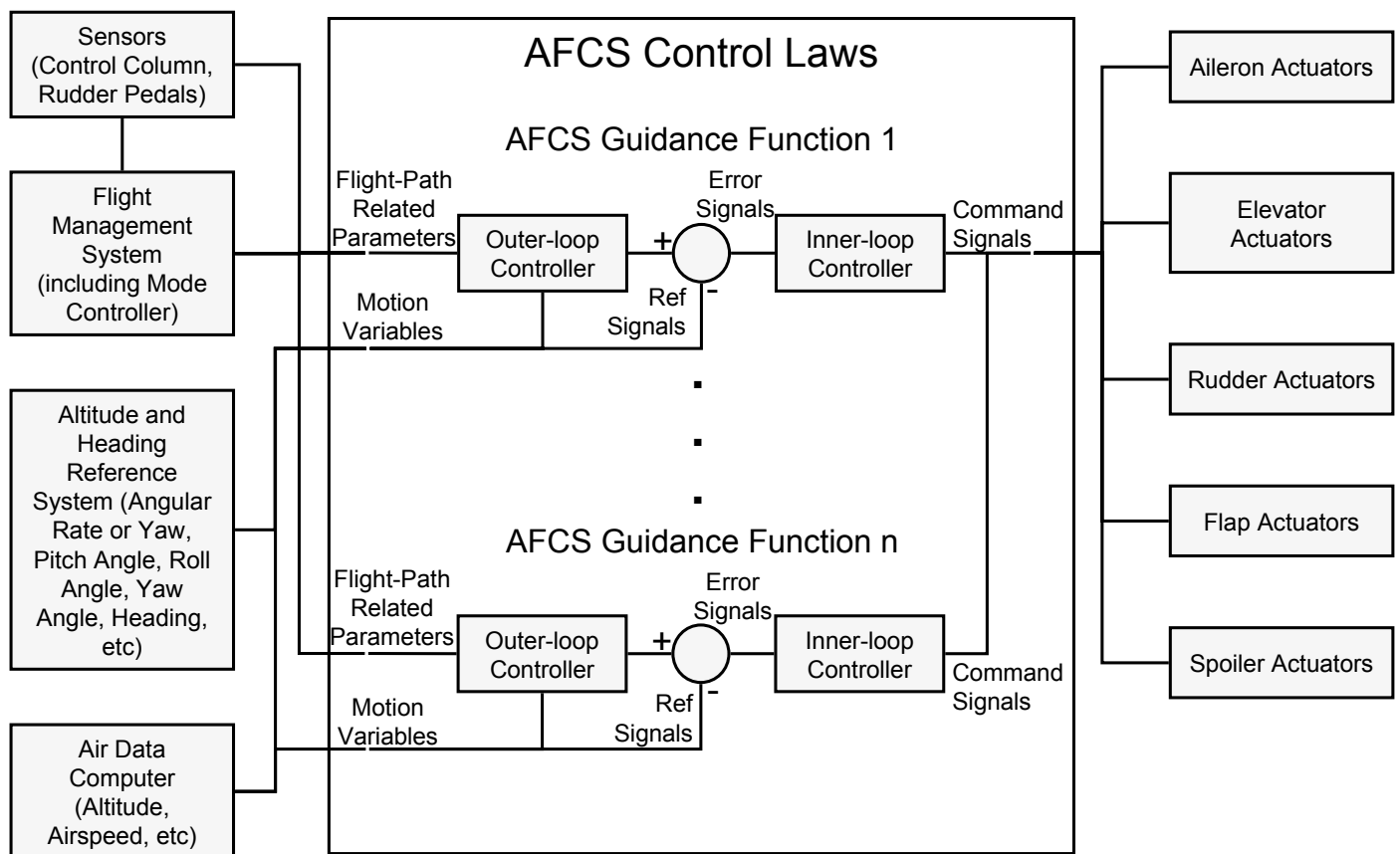


AFCS Guidance Functions





AFCS Software Functions



29

Directorate General Technical Airworthiness (DGTA-ADF)



Software Safety Analysis Techniques

- Software Functional Failure Analysis
- Software Fault Tree Analysis
- Software FMEA (FMECA)
- Software HAZOP (DEF STAN 00-58 Computer HAZOP)
- Software Hazard Analysis and Resolution in Design (SHARD) – refinement of software HAZOP
- Markov Analysis and Data Flow Diagrams
- Petri Net Analysis
- Software Sneak Analysis
- and many more...

30

Directorate General Technical Airworthiness (DGTA-ADF)





Analysis Approach

- One technique in isolation will not be sufficient.
- There is no technique that is suitable for all situations.
- A combination of techniques will need to be selected for a particular application.
 - There is no predefined or regulated solution.
 - Intelligent thought is required.
- We'll have a look at SHARD.



Software Hazard Analysis and Resolution in Design

- SHARD employs a series of guidewords to classify how the information flows and associated communication events (and associated services) might deviate from their intended forms.
- The guidewords are:
 - Omission – service not delivered.
 - Commission – service delivered when not required.
 - Early – service delivered, but early.
 - Late – service delivered, but late.
 - Value – service delivered, but with incorrect value.



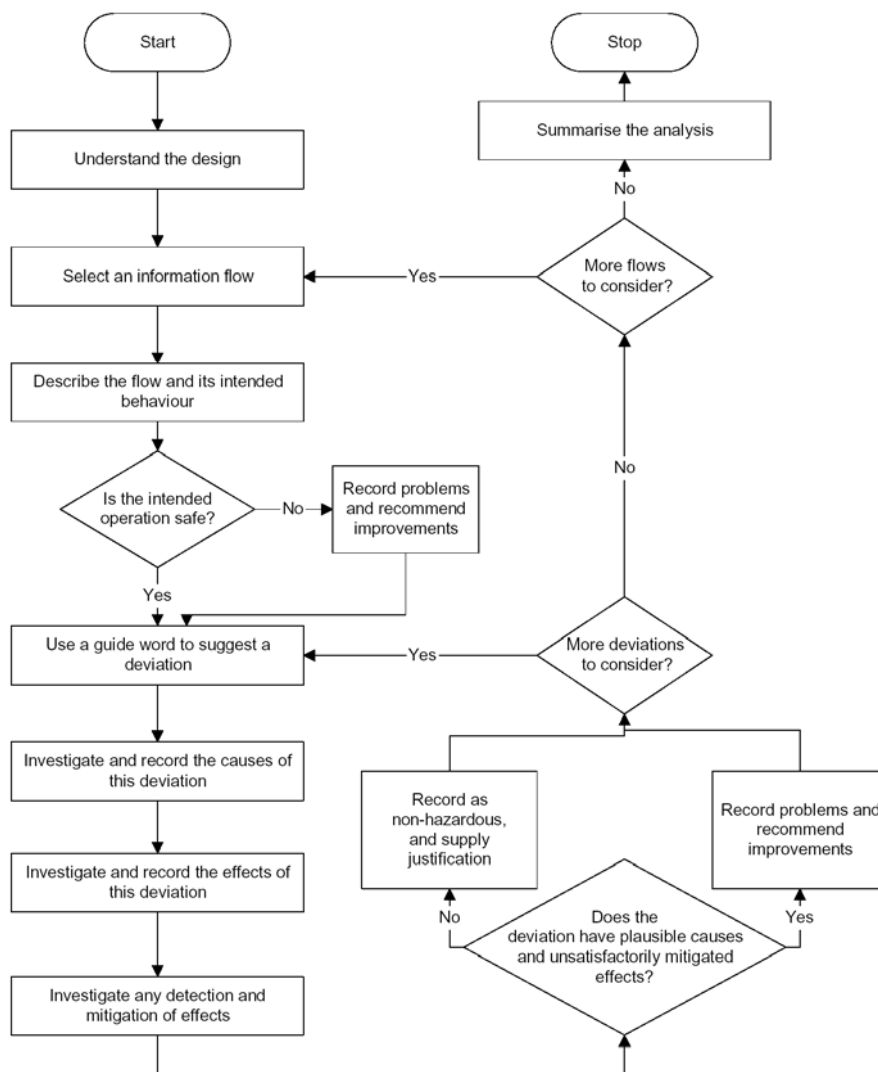


SHARD

- SHARD requires that the system be analysed “backwards” from the outputs (i.e. identify the system level effects first) back towards the inputs.
- The internal and input deviations are expressed in terms of how they cause or contribute to deviations in downstream items already investigated.
- Additional causes
 - Failure of the information flow itself
 - Failure of the flow source
 - Failure of the flow initiator

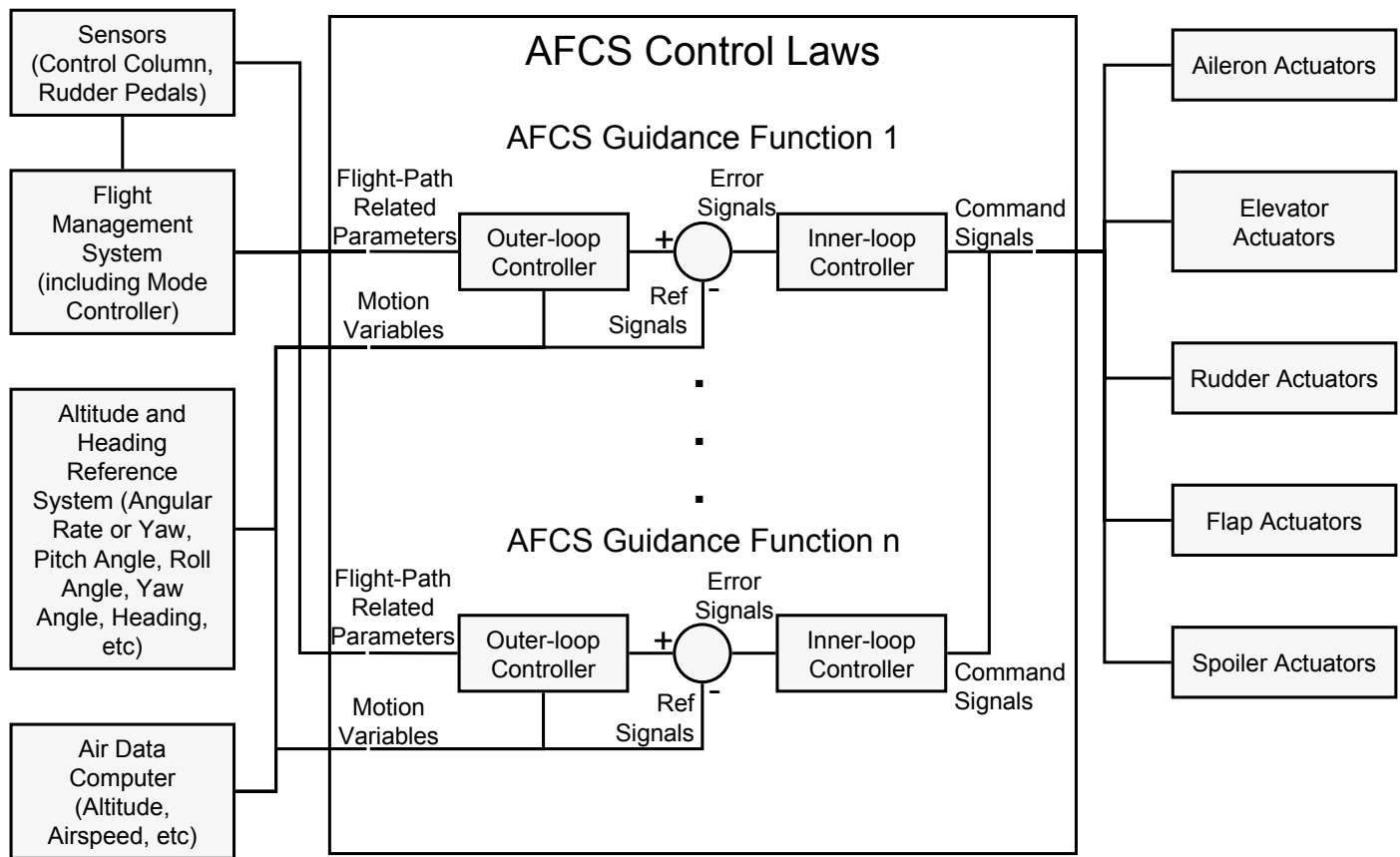


SHARD Process





AFCS Software Functions



Flight Control System SHARD – Command Signals

Guide Word	Deviation	Cause	Effect	Detection/Protection
Omission	Command signals not applied to actuators when required.	Programming error within Inner Loop Controller. Inner Loop Controller is not called.	Aircraft actuator states remain unaltered. Aircraft may fail to respond to pilot or FMS direction.	Inner Loop Controller to provide explicit output for each AFCS cycle. All exit points from Inner Loop Controller require an explicit update of command signals. AFCS cycles shall be explicitly synchronised to hard real time clock. Monitor to ensure Inner Loop Controller completely executes for each AFCS cycle.
Commission	Command signals applied to actuators when not required.	Programming error within the Inner Loop Controller. Invalid process writes command signals. Inner Loop Controller is called when not required.	Aircraft actuators move in unintended fashion. Aircraft responds incorrectly to pilot or FMS direction. Possible hardover or loss of control.	Inner Loop Controller shall be the only process to write to command signals. Inner Loop Controller shall only be permitted to write control signals once per AFCS cycle.
Early	Command signals applied to actuators earlier than required.	Programming error within the Inner Loop Controller. Inner Loop Controller is called earlier than required.	Aircraft actuator movement leads intended output, perhaps resulting in flight control oscillations (pilot or system induced).	Inner Loop Controller to provide explicit output for each AFCS cycle. AFCS cycles shall be explicitly synchronised to hard real time clock.
Late	Command signals applied to actuators later than required.	Programming error within the Inner Loop Controller. Inner Loop Controller is called when not required.	Aircraft actuator movement lags intended output, perhaps resulting in flight control oscillations (pilot or system induced).	Inner Loop Controller to provide explicit output for each AFCS cycle. AFCS cycles shall be explicitly synchronised to hard real time clock.
Value	Incorrect command signals applied to actuators.	Programming error within the Inner Loop Controller. Incorrect error signals passed to Inner Loop Controller.	Aircraft actuators may move in unintended fashion. Aircraft responds incorrectly to pilot or FMS direction. Possible hardover or loss of control.	Limit and reasonableness checks shall be performed on error signals passed to Inner Loop Controller. Limit and reasonableness checks to be performed on Inner Loop Controller outputs prior to writing command signals.





- Apply SHARD to other data flows working backwards towards the input.
 - Remember, each failure condition will now be expressed in terms of how they cause or contribute to deviations in downstream items already investigated.
- Continue to identify software safety requirements relating to detection and protection criteria.
 - Repetition of detection and protection criteria (and necessary software safety requirements) is OK – it is just establishing the importance of particular criteria.
- Present list of software safety requirements to System Safety Working Group to determine whether they are appropriate in the full system context.



Identify Generic Software Safety Requirements





Generic Software Safety Requirements

- Generic software safety requirements serve a number of purposes:
 - Implement Lessons Learnt
 - Generic software safety requirements often evolve from accidents
 - Treat Common Issues
 - Many software systems share the same susceptibilities
 - Not Constrained by Functionality
 - Many software faults have no direct relationship to functionality, so a function based assessment might not identify them
 - Check on Completeness of Software Hazard Analysis
 - Helps determine whether software hazard analysis has identified enough software safety requirements



Generic Software Safety Requirements

- System Design
- Power-Up System Initialisation
- Computer System Environment
- Self Check
- Safety Critical Computing System Function Protection
- Interface Design Requirements
- Human Interface
- Critical Timing and Interrupt Functions
- Software Design and Development
- Software Maintenance
- Software Analysis and Test

Source: JSSSC SSSH





Generic Software Safety Requirements

- Safety Kernel ROM
 - Safety kernels should be resident in non-volatile ROM or in protected memory that cannot be overwritten by the computing system.
- Inadvertent Jumps
 - The system shall detect inadvertent jumps within or into safety critical functions; return the system to a safe state, and, if practical, perform diagnostics and fault isolation to determine the cause of the inadvertent jump.
- Decision Statements
 - Decision statements in safety-critical computing system functions shall not rely on inputs of all ones or all zeroes, particularly when this information is obtained from external sensors.

Source: JSSSC SSSH



Leveson's Safeware Generic Software Safety Requirements

- State Completeness
- Input and Output Variable Completeness
- Trigger Event Completeness
 - Robustness Criteria
 - Non-Determinism
 - Value and Timing Assumptions
- Output Specification Completeness
 - Environmental Capacity Considerations
 - Data Age
 - Latency
- Output to Trigger Event Relationships
- Specification of Transitions Between States
 - Reachability
 - Recurrent Behaviour
 - Reversibility
 - Pre-emption
 - Path Robustness
- Constraint Analysis





- The system and software must start in a safe system state. Interlocks should be initialised or checked to be operational at system start-up, including start-up after temporarily overriding interlocks.
- All incoming values should be checked and a response specified in the event of an out of range or unexpected value.
- Safety critical outputs should be checked for reasonableness and for hazardous values and timing.
- Reachable hazardous states should be eliminated or, if that is not possible, their frequency and duration reduced.



Review Relevant Lists of Software Safety Requirements

- Review all relevant lists of software safety requirements and identify those that might be relevant to the system.
- This is as easy as stepping through the list with a highlighter in hand. Alternatively use a working group/team to conduct the activity.
- For example, for a flight control system as described in this presentation, a review of those generic requirements identified approximately 70 potentially relevant high level software safety requirements.





Verify that Software Satisfies Software Safety Requirements



Software Verification

- Once the software safety requirements have been identified it must also be shown that the software satisfies those requirements.
- Verification techniques can be difficult, expensive or time consuming to apply. Therefore it is important to understand what techniques are most appropriate to what aspects of safety critical systems.
- Techniques include:
 - Formal Methods
 - Mandated by DEF STAN 00-55 SIL 4
 - Model Checkers
 - Static Analysis
 - Dynamic Testing





Correctness by Construction

“To control the design and construction of a product to assure that requirements are satisfied.”

Static analysis, formal methods, etc.

Correctness Through Test

“To prove that a developed product satisfies requirements by dynamically exercising functionality.”

Black and White Box Testing



Static Analysis

- Analysis of source code before it is executed.
- Techniques include:
 - Data Flow Analysis, Control Flow Analysis
 - Symbolic Execution, Check of Source Against a Formal Mathematical Specification
 - Checking against a Language Subset/Coding Standards
- Advantage – if a property is shown to hold, it will hold for all scenarios.
- Disadvantage – some software coding architectures and techniques do not easily lend themselves to static analysis.
 - Recursion, pointers, etc
 - May be difficult to employ retrospectively
- Static analysis is heavily reliant on software tools.





Dynamic Testing

- Execution of the code against numerous test cases at varying levels of integration.
- Because software is complex, it is not possible to execute test cases for all possible inputs and outputs.
- Techniques to reduce number of test cases include:
 - Equivalence Classes
 - Boundary Value Analysis
 - Structural Testing (Statement Coverage, Decision Coverage, Modified Condition/Decision Coverage)
- Static Analysis and Dynamic Testing used effectively to complement each other provide deep insight into the behaviour of a software program.



Software Aspects of Safety Case

- **Implementation of System Safety Requirements**
 - Requirements to detect and handle other system failures or treat system level hazards.
- **Detect and Handle Input Failure Conditions**
 - Software can detect and handle bad inputs.
- **Detect and Handle Internal Failure Conditions**
 - Software can detect and handle internal failures without propagating to a system level failure.
- **Integrity**
 - Where a software function is relied upon above, sufficient proof exists that the software satisfies the requirements.
- **Completeness**
 - The system and software safety processes have completely considered the interaction between the software and the system.
 - The most difficult part: necessitates a structured approach.





A Real Life Case Study



51

Directorate General Technical Airworthiness (DGTA-ADF)



AFTI F-16

- Advanced Fighter Technology Integration (AFTI) F-16
 - triple redundant digital flight control system (DFCS)
 - analogue backup
- DFCS was an asynchronous design
 - channels run fairly independent of each other
 - each computer samples sensors independently
 - evaluates control laws independently, and
 - sends its actuator commands to an averaging component that drives the actuator concerned

52

Directorate General Technical Airworthiness (DGTA-ADF)





A Serious Shortcoming

- What happens when sensor noise and sampling skew cause independent channels to take different execution paths at decision points resulting in the production of widely divergent outputs?
- Occurred on Flight 44 of the AFTI F-16
 - Each channel declared the others failed
 - Analogue backup was not selected because the simultaneous failure of two channels had not been anticipated
 - Aircraft was flown home on a single digital channel
- Note now that all protective redundancy had been lost
 - yet no hardware failure had occurred
- Numerous other mishaps subsequently traced to the same issue



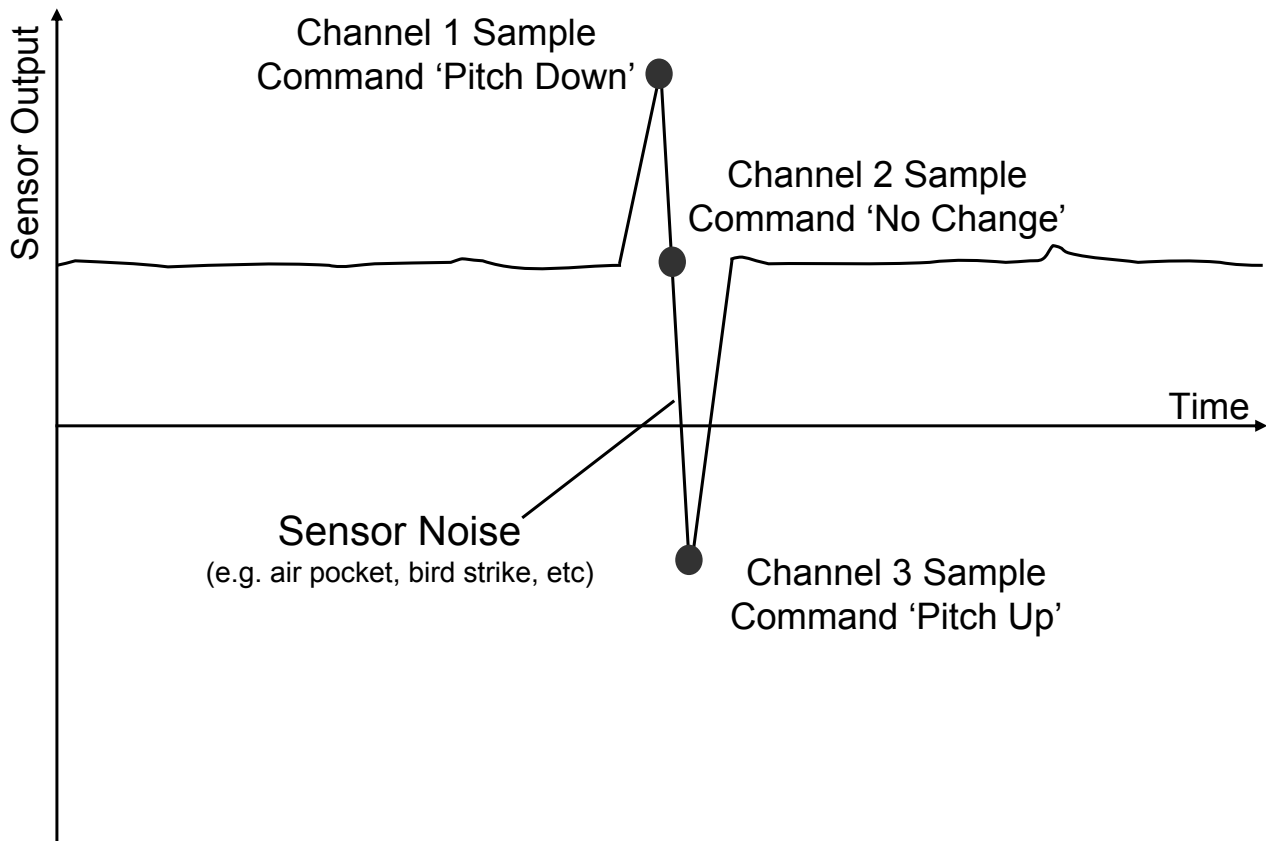
The Assessment

- The initial fix – introduce voting (as opposed to averaging), extensive simulation and testing performed.
- On the next flight – the problem was still there
- The final fix:
 - The asynchronous design, while introducing some independence between channels, exposed the software to common sensor failure modes.
 - Introduction of a requirement to synchronise the channels at decision points in the control laws.
 - Required a complete system redesign.
 - First discovered on Flight 44 – a long way into the design process.
 - Would have been cheaper to fix if detected earlier.





The Problem (Exaggerated)



Conclusion and Summary

- **System and Software Safety provide assurance that the following error and accident causes are eliminated:**
 - Discrepancies between documented requirements specifications and the requirements needed for the correct functioning of the system.
 - Misunderstandings about the software's interface with the rest of the system.
 - Requirements specify behaviour that is not safe from the system perspective.
 - Requirements do not specify some particular safety behaviour.
 - The software has unintended (and unsafe) behaviour beyond what is specified in the requirements.
- **Remember:**
 - The identification and allocation of software safety requirements are key to the realisation of acceptably safe software intensive systems.
 - Verification activities cannot contribute to safety if the requirements being verified do not specify safe behaviours.





Questions

