Xday, XX XXX 2015.

9.30 am - 11.15am ***(check this!)***

University of Glasgow

DEGREES OF BEng, BSc, MA, MA (SOCIAL SCIENCES).

COMPUTING SCIENCE - SINGLE AND COMBINED HONOURS
ELECTRONIC AND SOFTWARE ENGINEERING - HONOURS
SOFTWARE ENGINEERING - HONOURS

SAFETY-CRITICAL SYSTEMS DEVELOPMENT

Answer 3 of the 4 questions.

SCS Exam M

1.

a) Briefly describe the main benefits to be derived from accident investigations for the development of safety-critical software.

[5 Marks]

[Seen/Unseen problem]
Please note that many different answers are possible in most questions – I am looking for the quality of the argument not an enumerated list of pre-specified solutions. Hence, for some questions I have given more solutions than there are total marks. The sample solutions represent my best effort to answer the question; I would not expect such full answers under exam pressure. Students will not be penalized for incorrect answers hence I do not want to write "give four reasons…" – if they give 5 and 1 is incorrect then they may still obtain full marks. This will be explained in the course and in revision classes.

Accident investigations identify the causes and contributory factors leading to failures in safety-critical systems (1 mark). They provide learning opportunities; helping to ensure that accidents do not recur (1 mark). They also provide deeper insights beyond the immediate recommendations in a report, for instance when surveys look across patterns of previous adverse events (1 mark). They can also be used to inform subsequent risk assessment – indicating the past probability and consequences of adverse events (1 mark). In terms of software, accident investigations can provide insights into the development practices that led to a failure (1 mark). The focus should be on the underlying causes rather than particular bugs. Again, in practice, this may be hard to achieve as investigators often lack the technical background to find these causes (1 mark). Where they can call on external support they may not be able to trace back particular bugs through to problematic development practices (1 mark).

b) What factors make the investigation of software failure more difficult than other aspects of conventional accident investigation – for instance, compared to hardware or human factors?

[5 Marks]

[Unseen problem]
There are well-understood techniques for the analysis of hardware following major accidents (1 mark) – it is also possible to develop simulations and models with relatively deterministic behavior to explore and, ideally, recreate failure modes (1 mark). This is more problematic in the case of human factors – often there is a lack of video evidence and audio cockpit or cab recordings can tell what happened in terms of operator interaction but they seldom tell us why someone did something (1 mark). Fortunately, there are well-developed theories of cognition and perception that can be applied to derive insights into the factors that influence human behavior (1 mark). In software things are far more complicated. Firstly, we often lack the physical evidence that is available in the aftermath of hardware failures (1 mark). Logs are limited and even if we can identify major events/transactions it is often impossible to recreate the contents of volatile memory in the moments before an accident (1 mark). Further complications arise when we may have Commercial Off the Shelf and legacy software for which we either lack the source code (1 mark) or lack the technical insights necessary to determine what happened in a software failure (1 mark). We do not have the same theoretical models of software failure that could parallel those for human behavior (1 mark). Some techniques from software engineering

can be transferred – for example, use a lifecycle model to trace the underlying potential concerns during the development of safety-critical code but often there is limited evidence that we can rely on to support causal theories (1 mark).

c) The UK Health and Safety Executive argue that:

> "It is the potential consequences and the likelihood of the adverse event recurring that should determine the level of investigation, not simply the injury or ill health suffered on this occasion. For example: Is the harm likely to be serious? Is this likely to happen often? Similarly, the causes of a near miss can have great potential for causing injury and ill health. When making your decision, you must also consider the potential for learning lessons. For example if you have had a number of similar adverse events, it may be worth investigating, even if each single event is not worth investigating in isolation. It is best practice to investigate all adverse events which may affect the public".

Write a brief technical report describing how these more general arguments might be used to improve the development of software within safety-critical systems.

[10 Marks]

[Unseen problem]

There are many potential answers. One approach is to focus on the need not simply to wait for an accident but also to be more pro-active in considering near miss incidents to improve software engineering (1 mark). This goes beyond conventional bug reports. Most safety-critical systems, have a 5-6 week update cycle (1 mark). Verification and validation suites are used to ensure that the next build reaches a sufficient level of reliability before known bugs are corrected (1 mark). Too often, these bugs are corrected in a piecemeal way (1 mark). Companies do not pause to look for common factors that lead to the software problems that are addressed by these upgrades (1 mark). For instance, these might include flawed requirements (1 mark), poor communication with sub-contractors (1 mark), inadequate or confused guidance on programming standards/practices (1 mark), rushed deadlines and insufficient budgets (1 mark) etc. One of the most significant lessons to be drawn during the development of safety-critical systems is to achieve a clear understanding of the reasons why a potential bug is not identified during subsequent test cycles (1 mark). It is also important to identify and strengthen those processes that eventually did lead to a problem being located and addressed (1 mark). Other solutions might look to resilience engineering focusing on the good practices that DID identify potential failures rather than those rare situations where safety was threatened (1 mark).

2.

a) NASA Technical Standard 8719.13C stresses the importance of traceability in safety-critical systems:

> "Traceability is a link or definable relationship between two or more entities. Requirements are linked from their more general form (e.g., the system specification) to their more explicit form (e.g., subsystem specifications). They are also linked forward to the design, source code, and test cases. Because many software safety requirements are derived from the system safety analysis, risk assessments, or organizational, facility, vehicle, system specific generic hazards, these requirements; these will also be linked (traced) from those specific safety data products (examples include safety analysis, reliability analysis, etc.)".

Identify two problems with ensuring traceability in the development of complex, safety-critical systems and describe how to mitigate each of these problems.

[4 Marks]

[Seen/Unseen problem]
There are many potential answers to this question – problems include the complexity of the projects (1 mark), the use of sub-contractors (1 mark), the use of Commercial off the Shelf (COTS) code (1 mark), the presence of legacy code (1 mark) and the need to maintain programs for long periods of time when subsequent modifications may undermine the traceability that was achievable before (1 mark). Potential solutions include the use of programming codes/guides (1 mark), model based software development can also be used to automatically maintain constraints from high level designs through to partial or full implementations (1 mark). Safety cases may also play a role here – explaining key relationships between different development artifacts (1 mark).

b) The NASA Technical Standard does not follow the approach adopted in IEC 61508, instead NASA-STD-8719.13C states that the overall system safety analysis is used to guide the assessment of software risk and that the following particular factors need to be considered:

- Level of autonomy,
- system complexity,
- software size,
- hardware/software trade-offs.

Briefly explain why each of these factors might contribute to the likelihood and consequence of software related failures. Why can it be hard to identify appropriate metrics to measure these software risk factors?

[6 Marks]

[Unseen problem]
Answers should address each of the factors mentioned in the NASA Standard:

- *Level of autonomy increases the likelihood of software related incidents because operators may not be involved in the direct supervision of a safety-critical system hence they are arguably less likely to anticipate an incident either preventing it or mitigating the consequences (1 mark). Other concerns relate to the erosion of control skills should the operator have to intervene in the aftermath of a failure involving autonomous systems (1 mark).*
- *system complexity is an obvious factor in the risks associated with safety-critical software. Complexity can be broken down into more detailed factors including the degree of interaction and integration across components (1 mark). Answers might also consider the organisational complexity associated with the different groups who must cooperate to ensure safety across complex software systems (1 mark).*
- *software size, the number of source lines of code is a key issue. This is also related to the choice of language given that we must consider errors in the compiler/interpreter for high criticality systems (1 mark). In the course we have looked at the Musa expansion ration for higher level languages vs machine code. He would argue that higher level languages increase the probability of errors from compilation – others would argue that the probability of error increases because of the problems in understanding/maintaining lower level coding structures (1 mark).*
- *hardware/software trade-offs. In general, hardware is considerably easier to reason about than software – given the need to ensure correctness across different levels of virtualisation (1 mark). However, the increasing use of FPGAs and 'smarts' blurs some of these traditional distinctions (1 mark).*

It is hard to identify appropriate metrics to measure these software risk factors because there are no objective, agreed metrics for the level of autonomy. End-user engagement with the underlying application may well change over time – in the course we discuss autonomous cars where it is often assumed that the driver retained overall responsibility for safety and must intervene if necessary. Over time, reliable systems would make it increasingly unlikely that drivers might maintain a necessary level of vigilance but how to measure this? (1 mark). Similarly, complexity is a thorny issue in computing science going back to McCabe and Cyclomatic complexity. The size of code is hard to measure – Source Lines of Code (SLOCs) are superficial given the conflict between low and high level languages mentioned above – is machine code better than say Ada in terms of the number of errors that we might anticipate? (1 mark) As before, it is almost impossible to agree on a number to indicate the risk caused by substituting software for hardware – especially given that the distinctions are no longer clear between what is hardware and what is software.

c) NASA-STD-8719.13C identifies the importance of links between risk analysis and verification and validation:

> "…software safety testing will include verification and validation that the implemented fault and failure modes detection and recovery works as derived from the safety analyses, such as PHAs, sub-system hazard analyses, failure-modes-effects-analysis, fault-tree-analysis. This can include both software and hardware failures, interface failures, or multiple concurrent hardware failures. Fault detection, isolation and recovery (FDIR) is often used in place of failure detection when using software as software can detect and react to faults before they become failures".

Explain the problems that can arise when trying to maintain the links between risk analysis and verification and validation.

(Hint: you can use your answers to the previous parts of this question to support your solution]

[10 Marks]

[Unseen problem]
Risk assessment is difficult for most safety-critical systems because we seldom have the data from previous systems that we can use to guide our assessments of consequence and likelihood for the identified hazards (1 mark). Instead, we may have to look to previous generations of systems or to similar applications in different countries or industries to guide our assessments (1 mark). There will usually be huge differences in the mode of operation or in the context of use which means we need to rely on expert judgment and subjectivity when we think of the risk reduction implied in Safety Integrity Levels that are used – for example in software development under 61508 (1 mark).
Verification and validation activities can be used to ensure that our risk assessment has been sufficiently robust (1 mark). For example, we might begin to deploy experimental systems in controlled environments or with careful supervision (1 mark). This can reveal unanticipated hazards. It can also help to identify situations where we might have underestimated the probability of certain adverse events or environmental conditions. Hence, we need a link between risk assessment and the V&V activities that would help us to update our initial assumptions as we learn more about the particular circumstances of our systems (1 mark).
We can use the previous questions to talk about some of the problems that can arise when maintaining these links. The scale of the system (from part b) (1 mark) might make it hard to trace (from part a) (1 mark) the relationship between a hazard that has been identified and the lower level, more detailed V&V activities that are intended to determine whether the system is robust to that potential threat. Similarly, we could argue that if part of the system is provided on a service level contract from a sub then it may be difficult to ensure that the results of a V&V activity are communicated (1 mark) across organizational boundaries to provide feedback on a hazard analysis (1 mark).

3.

a) The Boeing 777 airplane required some 2.5 million lines of newly developed software. If we include Commercial Off The Shelf (COTS) and optional software this rises to more than 4 million lines of code, across 79 sub-systems from suppliers in many different countries. Identify factors that complicate the safety-critical software engineering of such systems.

[4 Marks]

[Unseen problem]
This is a very open question so there is lots of scope for more able students. Some of these factors can be taken from the previous question eg scale (1 mark), complexity (1 mark). The key is to link these ideas to the information provided about the 777 – so solutions might comment on the choice of high vs low-level languages (1 mark) used in development of these systems, other answers might look instead at the IP barriers created by the use of COTS (1 mark). This makes it hard to do white box testing (1 mark) and prevents the usual traceability techniques that might be expected with bespoke software (1 mark).

b) Boeing conducted a post-development review of the 777:

"We found no correlation between the language used and the number of problems found in the system. We found instances where Ada was used effectively and the developers felt it substantially reduced software integration problems. In other cases development was hampered by problems with compilers and with other support tools. Many suppliers chose to use a restricted subset of Ada which led to fewer problems but lesser benefits".

Use this quote to justify the properties that you would look for when selecting an appropriate compiler for use in the development of safety-critical software.

[6 Marks]

[Unseen problem]
The quote suggests that there are no simple conclusions when selecting particular programming languages. The use of Ada is intended to reduce errors – for instance, by the enforcement of strict typing (1 mark) and the provision of extensive support for exception handling (1 mark). However, Boeing seem to indicate that this did not significantly reduce errors.

Ideally we might look for a language that prevents programmers from bad practice (eg eliminating support for arbitrary jumps using labels/gotos, restricted use of the heap, limits on cacheing etc) (1 mark). Boeing identify a Catch-22 concern when errors are reduced by removing features of the higher level language (for example preventing the use of dynamic memory allocation) at the expense of the usability/productivity of the language (1 mark).

The observation that there were problems with 'compilers and with other support tools' is particularly interesting – Ada compilers are supposed to meet standards specified by the US DoD and yet there still seem to be issues (1 mark). The aim was to ensure that the compilers supporting higher-level languages were reliable enough to avoid the need to use machine code (with the associated concerns over cost/maintenance for lower level languages) (1 mark). Problems with tool support are less surprising – compilers for more popular higher level

languages tend to be supported by an array of development environments but these seldom have the degree of care in terms of the coding that was devoted to the original compiler development (1 mark).

c) The FAA recently issued an airworthiness directive that required a software update for the 777's three autopilot flight director computers. Pilots noticed unexpected resistance when interacting with the control column, causing them to abort the take-off at high speed. Boeing identified two possible causes. Pilots could mistakenly press the autopilot switch on the aircraft's mode control panel (MCP) when attempting to engage the auto-throttle. Alternatively, pilots may mistakenly press the left autopilot switch on the MCP due to previous training in Boeing 757, 767, 747-400 models. These were rare events; in 15 years and more than 4.8 million flights there were nine reported instances of a rejected take-off because of inadvertent engagement of the autopilot without any runway overruns or injuries.

   What lessons can we draw for the future development and operation of safety-critical software from these 777 autopilot concerns?

[10 Marks]

[Unseen problem]
The first major lesson is that software failures can be intermittent and rare events (1 mark). As mentioned "in 15 year and more than 4.8 million flights there were nine reported instances of a rejected take-off". However, the potential consequences of such rare events justify efforts to address the causes of the problem. Unfortunately, given the complexity of modern software (4 million lines of code mentioned in part a and many sub-contractors) it can be difficult to identify the causes of these rare events (1 mark). This is illustrated by the way in which the manufacturer has two causal hypotheses (1 mark). In the course, we have mentioned Dijkstra's maxim – testing proves the presence of bugs and not their absence – in consequence we may have other failure modes that are not recommendation in this directive from Boeing (1 mark).
Other answers might focus on the software updating process and link it to previous answers in the paper – for example, talking about traceability across the update process (1 mark). Solutions could mention concerns over V&V for initial safety properties to ensure that the updates do not undermine any non-functional requirements met by the initial implementation (1 mark). It is also possible to focus on the role of redundancy in the flight director computers. It does not matter how much diversity/redundancy one employs if all of the systems receive the same incorrect input values (1 mark). Another line of argument might focus on the human factor issues – in particular the transfer effects of skills gained on earlier systems then being inappropriately transferred to a new system, even when they are built by the same manufacturer (1 mark). Other answers could look at the role of human error when pilots might 'mistakenly press the left autopilot switch on the MCP' (1 mark). This does not identify the cause of the error (1 mark), however, training might be an issue as before – or the design/layout of the cockpit (1 mark).

4.

Write a technical report for the senior management of a safety-critical software company explaining the main causes of regulatory lag in safety-critical industries. Use this analysis to identify ways in which the company might continue to develop and market safety-critical software in areas that suffer from regulatory lag.

[20 Marks]

[Essay]

As in previous questions there are many different solutions here with ample space for more able students to focus on particular examples that we have met during the course.

Regulatory lag describes situations in which companies seek to innovate at a rate that is not matched by guidance on acceptable means of conformance provided by the regulatory authority (1 mark). This could be illustrated by confusion over the operation of RPAS in controlled air space (1 mark) or the degree of cyber-security protection that should be provided by the operators of national critical infrastructures (1 mark). Regulatory lag causes concern because it can prevent a new industry from growing (1 mark). Companies will be reluctant to invest because they cannot determine what they might need to do to satisfy the regulator (1 mark) – they might be prevented from operating or if they can operate the lack of definitive regulatory guidance might create scope for litigation should an accident occur (1 mark). From the regulators point of view, they often lack leading edge technical skills with limited salaries and stringent fiscal constraints (1 mark). They may also be reluctant to issue inappropriate guidance before an industry is mature and the risks are better understood (1 mark).

There are significant differences across Europe and North America – in the USA, there is a tendency to offer waivers to other regulations allowing industry to develop and gathering the insights sufficient to draft new regulations. In Europe, there is a tendency to extend existing regulations but after some period of time – hence creating uncertainty (1 mark). In the course we discussed the regulations covering sub-orbital flights as an example of this (1 mark).

These two approaches can be contrasted in the second part of the question, for example through a case study, although this is not the only approach to the question. The developers of autonomous ars in both the USA and the UK, have permission for limited field trials. This can be thought of as a variant on the waiver approach (1 mark). There is no blanket approval for the introduction of autonomous cars but certain exceptions to existing regulations are allowed to enable the industry to develop (1 mark). An alternative approach would be to require that all autonomous systems have to reach the level of reliability exhibited by a human driver (meeting existing regulations) before it can be deployed. This might hold the industry back for many years (1 mark).

From a company perspective, one approach might be to develop the software even without explicit regulatory approval. This carries considerable risk if the regulator subsequently intervenes or if there is subsequent litigation showing the violation of best practice in conventional safety critical software development.

Alternatively, the company might seek regulatory approval through the provision of a waiver – this provides a degree of assurance (against litigation) but is hardly likely to encourage

customers who must be informed of the special 'exemptions' enabling the operation of the software.

Alternatively, the company might develop innovative software within the constraints of existing regulation. For example, ISO 26262 limits the use of AI in high-criticality automotive systems. This would prevent the use of self-modifying/learning algorithms in the on-board vision and sensing systems. Companies resolve this potential impasse by stating that the driver must retain ultimate responsibility for the safety of an autonomous system, even though the justification for autonomous software is to reduce the need for continual attention on driving tasks.