CS_313 High Integrity Systems/ CS_M13 Critical Systems

Course Notes

Chapter 5: The Development Cycle for Safety-Critical Systems

Anton Setzer
Dept. of Computer Science, Swansea University

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

December 2, 2011

(a) Life Cycle Models

- Critical systems have to be developed up to highest standards.
  - This means that one has to use methods which guarantee such standards.
  - The development of critical systems has to be well-documented and therefore the development process is much more formalistic the usual.
  - This is especially important since critical systems have often do be certified.
  - During certification, the documents used will be carefully checked.

Life Cycle Models

- Specification and verification are much more important than for ordinary software.
- The standard life cycle model used for critical systems is the **V-model**, which is very close to the waterfall model.
    - The V-model was developed independently simultanteously in Germany by a company in cooperation with the German Ministry of Defence, and by the National Council on Systems Engineering for satellite systems involving hardware, software and human interaction.

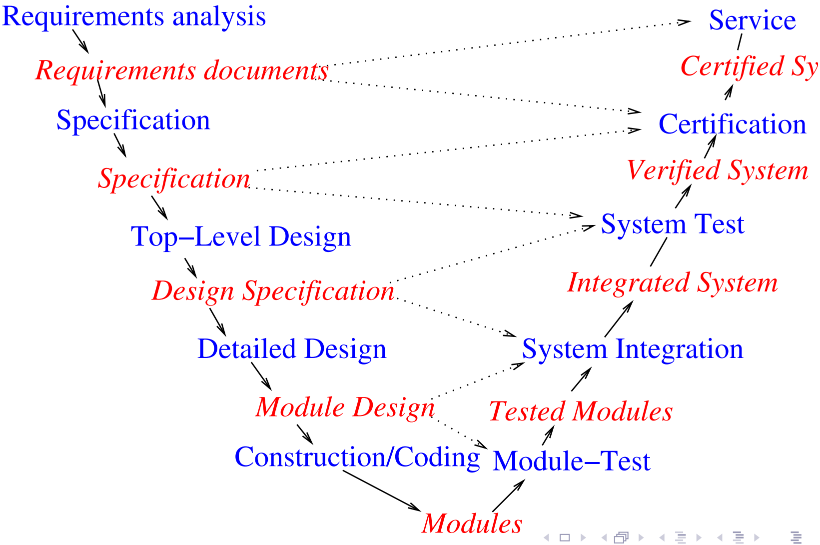
Life Cycle Models

- The origins of the V-model explain its tbfsuitability for critical systems, since **military software** are often **safety critical** and **satellite systems** are **mission critical**.
- Therefore that model was probably developed taking this into account.

Explanation Next Slide

- Items in blue Roman are development phases.
- Items in *red italics* are output from the development phases.
- $\longrightarrow$ is the primary flow of information.
- $\dashrightarrow$ is the secondary flow of information.

V-Development Life Cycle



Model from IEC 1508

The IEC 1508 model can be found in the Additional Material which is available from the website.

Material Moved to Additional Material

The material for this subsection has been moved to the additional material, which is available from the website.

Material Moved to Additional Material

Here we present only the material on Specifications. The rest has been moved to Additional material, which is available from the website.

Specification

- The goal of a **specification** is to define in an unambiguous manner, the precise operation of a system.
- Includes:
  - the functionality and performance of the system,
  - its interaction with other systems,
  - safety invariants of the system,
  - constraints of safety invariants on the design.
- In case of subcontracting of software, the specification forms a contract between the supplier and the customer.

Specification

- An ideal specification should be
    - correct,
    - complete,
    - consistent,
    - unambiguous.
- Especially completeness is often underestimated.

Example of an Incomplete Specification

- A carriage moves vertically along a guideway between two end stops.
- On each end-stop is a limit switch that should prevent further travel.
- If **neither limit-switch is closed** the system should allow the carriage to move in either direction under the control of other routines.
- If **the upper limit switch is closed** the system controlling the carriage should ensure that it can only move downwards and hence away from that end-stop.
- If **the lower limit switch is closed** the system controlling the carriage should ensure that it can only move upwards.

Example of an Incomplete Specification

- Missing: what happens if both switches are closed?
    - Could not happen if switches operate correctly.
    - However one switch might be broken, and then the system should deal with this error.
- As it stands, in this case the system might reach an unsafe state.

Problems of Natural Specifications

- Most specification are written in a natural language (e.g. English).
- There are 3 problems with natural language specifications.

1. Natural language is often **ambiguous**.
    - **Example:** "This toilet is available to disabled students and staff only".
        - Is it available to disabled staff only or to all staff?

Problems of Natural Specifications

2. Natural language specifications are much **longer** than mathematical formulations, and therefore it is more easy to **overlook** something.
    - That's the reason why in mathematics one writes formulae
        - e.g.

$$\forall x, y . x = y \rightarrow y = x$$

    - instead of natural language texts
        - e.g.

"for all x and y, if x is equal to y then y is equal to x"

Problems of Natural Specifications

2. (Cont.)
   - Without it would be much more difficult to keep an overview of what is currently available in a mathematical proof.
   - Similarly in natural language specifications one might insert **inconsistencies** or **inaccuracies**, which one would see immediately when using formal languages.
3. One cannot apply **automatic checks** (e.g. whether there are inconsistencies) to specifications written in natural languages.

Specification Languages

- Therefore **formal specification languages** have been developed.
  - Are used in industry.
  - Usually some **tool support** exists (syntax checks, some consistency checks).

Formal Specification Languages

- Two approaches:
  - **Model-based specification languages**:
    - Based on a general model for representing programs (usually a set theoretic model)
    - The system to be specified is constructed in this model using mathematical constructs such as sets and sequences.
    - The system operations are defined by how they modify the system state.

Formal Specification Languages

- **Algebraic specification languages**:
  - Systems are described in terms of operations and their relationship.
  - Relationships are described axiomatically.
  - With a consistent specification usually a large variety of models is associated.
  - The consequences of a specification are what holds in all models associated with a specification.

Formal Specification Languages

- Examples of formal specification languages:
  - **Algebraic languages:**
    - Sequential: Larch, OBJ, Maude, CASL
    - Concurrent: Lotos.
  - **Model-based languages:**
    - Sequential: VDM, Z, B-method, Event-B.
    - Concurrent: CSP, CCS, Petri Nets.
- Prof. Mosses was the leader of the initiative creating CASL.
- Dr. Roggenbach is a specialist on CASL, and has integrated CSP into it.
- Prof. Moller is a specialist on CCS.
- Prof. Tucker is a specialist on algebraic specification.
- Dr. Seisenberger, Dr. Harman are using and teaching Maude.
- Dr. Setzer is a user of CASL.

(d) Designing for Safety

We present here only some material. The rest has been moved to Additional material, which is available from the website.

Software Partitioning

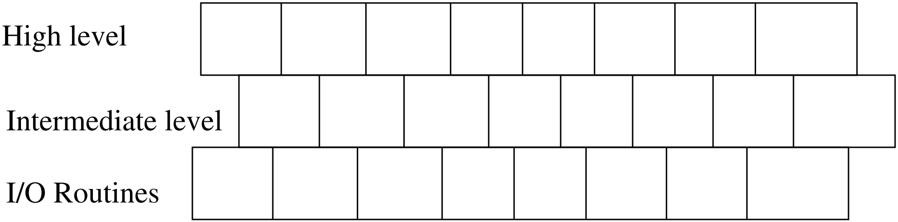
- Reason for partitioning of software:
  - Small units are **easier to understand** than a large monolithic program.
  - Partitioning provides **isolation** between software functions.
  - Allows to design the program so that **faults are contained** in one modules.
    - Makes **fault tolerance** possible.
  - Allows to assign to modules **different levels of integrity**.
    - If modules depend on each other, their criticality is that of the most critical one.
    - If modules are independent on each other, different (and often lower) levels of criticality can be assigned to them.

Hierarchical Design

- One approach towards designing systems is **hierarchical design**.
  - In a hierarchical design, a system is divided into a series of layers.
  - Modules within the higher layers depend for their correct operation on the correct functioning of lower-level components.
  - Lower levels might represent processors, control devices, sensors.
  - Higher levels might represent application-level software.
  - Intermediate levels are components like communication software and device drivers.

Layered Structures

- The result of a hierarchical design is a structure as follows:



High level

Intermediate level

I/O Routines

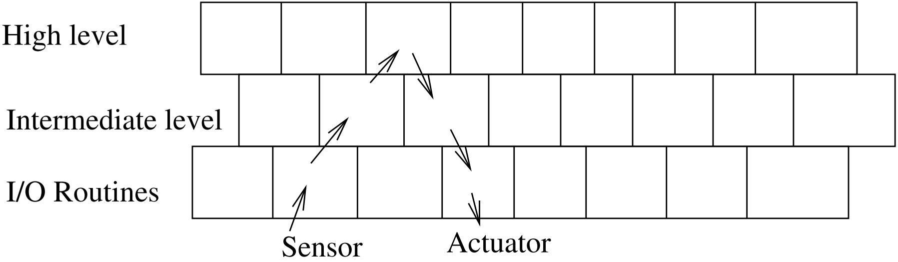
Layered Structures

- In a layered structure as before, upper modules depend on lower ones.
  - Therefore information about faults detected at lower levels have to be passed on to higher levels.
  - This is necessary in order to have good **fault management**, with the goal of having fault avoidance and fault removal.

Isolating Critical Functions

- It's important that critical functions are contained within modules, preferably within lower level modules.
- For instance, if a high level modules decides depending on information from one lower level module, whether a critical actuator controlled by another lower level module is activated, then this high level module and all intermediate modules involved have a high degree of criticality.

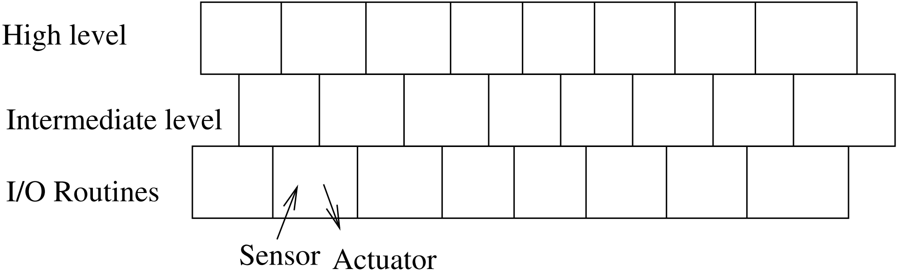
Long Chain of Responsibility



Better Architecture

- If instead this decision is done directly by one low level module, then only this small module is critical.
    - And it is much easier to verify a smaller module, rather than a big chain of modules.

Short Chain of Responsibility



Firewalls

- In critical systems, a **firewall** is a system which protects the critical elements of the system.
- A firewall might be
  - a **physical barrier**,
  - or a **logical barrier** to the system software, which prevents failure of the software outside the firewall from affecting the critical software within.
- Part of this is the **prevention of unauthorised access** or modification of data and code within the protected region.
  - That aspect of firewalls is what is associated with firewalls in the area of Internet security.

Safety Kernel

- A **safety kernel** is
  - a relatively small simple arrangement,
  - usually a combination of hardware and software,
  - that performs a set of safety-critical functions or provide operating system components that perform critical tasks.
  - Therefore the criticality of the system is concentrated in this kernel.
  - It is crucial that the kernel is well protected from outside influences.
    - Might be achieved physically, by use of separate hardware.
    - Might be achieved by software, by providing software isolation.

Example: Railway Control System

- For instance in a railway control system, one might have
  - A **small safety kernel**, which
    - receives high level commands about routes of trains to be chosen,
    - checks whether there are any conflicts,
    - and, if there are no conflicts, sets signals and activates switches accordingly.
  - A very **complex software**, which in an intelligent way controls the railway system
    - but all the commands of which are passed on to the small critical module.
  - Then one can assign a low level of integrity to the complex software, and only needs to assign a high level to the small safety kernel.

Software Isolation

- A unit in a program is **isolated**, if other modules can only influence it by using the public interface of the unit (which includes global variables).
- This means that
  - neither any local variable can be changed by any other unit,
    - access to the unit is only possible through the "front door",
  - nor the execution of the unit can be blocked by other modules consuming all the time or memory available.

(e) Human Factors in Safety

- As operators or users, human beings can be considered as **components of critical systems**.
- Humans bring both complications and potential benefits to a system.
  - **Complications:**
    - Humans are often **unreliable** and **unpredictable**.
    - Therefore many accidents are attributed to human error.
    - **Computers** are **superior** in terms of **speed** and the ability to **follow a predefined set of instructions**.

Benefits of Humans in Critical Systems

- **Benefits:**
  - Humans are **flexible** and **adaptable**.
  - They are extremely good at **dealing with unexpected events**.
  - They are invaluable if a system **strays from its normal operating regime**.

Liveware

- Humans considered as a further component in a critical system, implement safety features.
  - E.g. a pilot, which in an emergency takes over control over the plane provides some kind of **fault tolerance**.
- Therefore, one can apply the terminology **liveware** to humans as components.
  - Besides hardware and software, safety features can be implemented by liveware.
- Appropriate partitioning of safety features between hardware, software and liveware is important.

Role of Liveware in Critical Systems

- Because of their adaptability, humans form some kind of **backup system** in critical systems.
  - In order to make this possible, it is necessary that the human operators can **take over responsibility** from the computer system.
    - For instance, in an aircraft the pilot is allowed to override the automatic landing system, by switching to manual control.
    - Therefore the pilot can make mistakes the computer system would avoid.
    - But this allows the pilot to overcome faults within the system.
- In general this means that humans can be used very well in order to provide additional **fault tolerance**.

Problems of Liveware

- Problem is that humans add **complexity** to a system.
- Humans are **not as reliable** as a computer system, when it is about performing routine tasks.
    - Therefore one usually attempts to remove humans from tasks that can be implemented by following a **well-defined set of rules**.
- From the above considerations it follows that one preferably should
    - remove humans from routine tasks,
    - but use them in the form of controllers, which take over responsibility in case of an emergency.

Human Error

- When an accident occurs, the reasons will in most cases be attributed to either **system failure** or **human error**.
- Very often the conclusion is **human error**.
- However, many human errors are due to to **deficits in the Human-Computer Interface (HCI)**.
  - **Example:** If an air plane crashes because the pilot does not notice that it is short of fuel this is human error.
  - If that happens several times, then one can question the display and warning system of the aircraft, and therefore the **HCI**.

Material Moved to Additional Material

The material for this subsection has been moved to the additional material, which is available from the website.

Material Moved to Additional Material

The material for this subsection has been moved to the additional material, which is available from the website.

Material Moved to Additional Material

The material for this subsection has been moved to the additional material, which is available from the website.