Safety and Reliability of Embedded Systems

(Sicherheit und Zuverlässigkeit eingebetteter Systeme)

Introduction

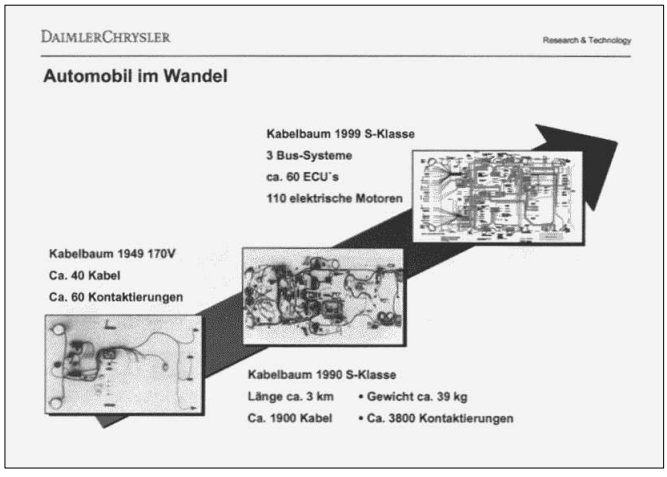
Content

- Paradigm-change in the automotive business
- Ariane 5
- Therac-25
- Definition of our research topic and focus
- Situation analysis of software development in practice
- Consequences
- Summary of available techniques

Motivation

- The majority of microprocessors is installed in technical (embedded) systems (varying statements on the number, but surely more than 90%)
- Many of these systems are not safety-critical (e.g. cellular phone)
- Others are safety-critical
  - Aircrafts
  - Trains
  - Cars
  - Medical equipment
  - ...
- Reliability and availability are always important

Motivation
Paradigm-change in the automotive business



Motivation
Ariane 5

June 4., 1996, Kourou / Fr. Guyana:
Maiden flight of the Ariane 5

```
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
  ...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get(vertical_veloc_sensor);
    sensor_get(horizontal_veloc_sensor);
    vertical_veloc_bias := integer(vertical_veloc_sensor);
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
end irs2;
```


Motivation
Ariane 5

- Cause
  - 37 sec. after engine start (30 sec. after liftoff) Ariane 5 had a horizontal velocity of 32768.0 (internal units). The integer conversion of the 64-bit floating point variable caused a data overflow. The second flight controller experienced the same problem 72 msec before and thus was not operational at that time. Diagnosis data were propagated to the main flight computer. These data were interpreted as valid flight data. Incorrect steering commands were sent. These caused a mechanical overload and finally Ariane 501 exploded.
- Effect
  - Total financial loss of 850 Million Euro

Motivation
Therac-25

- Therac-25 was a linear accelerator released in 1982 for cancer treatment by emitting limited doses of radiation
- This new model was software-controlled as opposed to hardware-controlled; previous units had software merely for convenience
- Controlled by a PDP-11 computer; software controlled safety
- In case of error, the software was designed to prevent harmful effects
- However, in case of software error, cryptic codes were given back to the operator: “MALFUNCTION xx”, where 1 < xx < 64
- Operators were rendered insensitive to the errors; they happened often, and they were told it was impossible to overdose a patient
- However, from 1985-1987, six people received massive overdoses of radiation; at least three of them died

Motivation
Therac-25

- Main cause
  - Race condition often happened when operator entered data quickly, then hit the UP arrow key to correct, and values weren’t reset properly
  - AECL (the company) never noticed quick data-entry – their people didn’t do this on a daily basis
  - Apparently the problem existed in previous units, but they had a hardware interlock mechanism to prevent it; here, they trusted the software and took out the hardware interlock

Motivation
Lessons from Therac-25

- Overconfidence in software, especially for embedded systems
- Reliability is not equal to safety
- No defensive design, bizarre error messages
- They just “bugfixed”, and didn’t look for root causes
- Improper software engineering practices
  - Most testing, in reality, was done in a simulated environment *and* a complete unit; little if any unit and software testing
  - They claimed 2700 hours of testing; it was really 2700 hours “of use”
  - Overly complex, poorly organized design
  - Blind software reuse

Motivation

- It is difficult to develop large, complex software and to guarantee that this software does not cause problems during operation
- If problems occur, these may cause catastrophic effects in technical application domains

Our research topic:
Software Engineering for Technical Applications

Focus:
Quality Management and Quality Assurance, i.e.,
Safety, Reliability, Availability and Real-Time Behavior
of Critical Software-Based Systems
(e.g. Transportation, Medical Systems, Industrial Automation)

Situation Analysis of Software Development in Practice

- Question: Who ensures that system development is perfectly done?
- Answer: Nobody!
- Consequence: The development is not complete with the implementation. Quality assurance is needed.
- Typical approaches
  - Ensure that the development processes are suitable

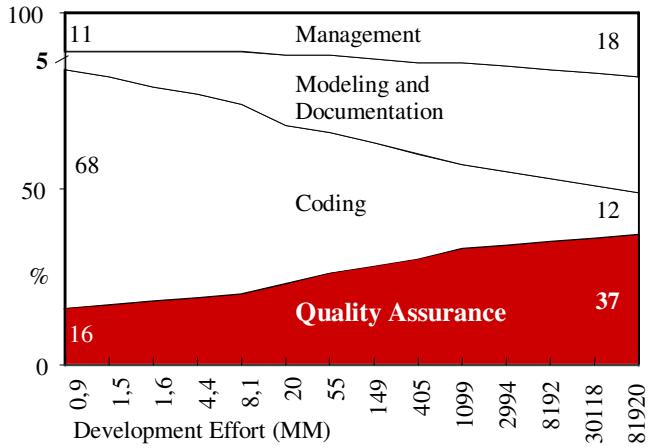
    => Quality management
  - Ensure that the development steps provided the desired results

    => Quality assurance (can also be done more or less formally and in a quantified or non-quantified manner)

Situation Analysis of Software Development in Practice

- According to M. Cusumano the defect rate of software shows the following trend (defects in 1000 lines of source code)
  - 1977: on average 7- 20 defects
  - 1994: on average 0,2 - 0,05 defects
  - In 13 years the defect rate could be lowered about 100 fold (but the size of software products increased)

Situation Analysis of Software Development in Practice
Increasing Importance of Quality Assurance



According to data from:
Jones C., *Applied software measurement*, New York: McGraw-Hill 1991

Consequences

- Software (and systems) quality has to be assured
  - Evaluation, validation and improvement of development processes
  - Accompanying quality assurance during the early development phases
  - Testing of the implemented software (the code)
- The software is large => several test phases are required

Consequences

- Highly varying demands on software (*experimental prototype* up to *engine control of a commercial aircraft*) => need of different methods between „trial" and „proof"
- It is not possible to guarantee that code is fault-free => it is required to determine the residual risks => **quantitative analysis methods**

Available techniques

- Modeling techniques
  - FMEA, FMECA: Identification of critical functions, blocks, modules, …; no real quantified results
  - Reliability block diagrams: Quantified results on reliability; not really applicable to software
  - Fault trees: Formal technique based on boolean logic and statistics; quantified results
  - Markov analysis and stochastic Petri nets: Formal technique (augmented state machines), quantified results
- Analytical techniques
  - Simulation, testing: Incomplete, no dependable results
  - Stochastic analysis: Commonly used for hardware, no widespread use for software
  - Formal verification: Complete (for certain fault-classes), but complicated