Embedded Real-Time Systems—Lecture 16

Prof. Dr. Reinhard von Hanxleden

Christian-Albrechts Universität Kiel
Department of Computer Science
Real-Time Systems and Embedded Systems Group

28 June 2011

Last compiled: July 4, 2011, 18:29 hrs

Introduction to Dependability

The 5-Minute Review Session

1. What are Büchi automata?
2. How can we check a model for liveness properties?
3. What is the WCET problem, why is it important?
4. What are the components of execution time analysis?
5. How can we measure execution times?

What's so Difficult About RT-Systems?

- Concurrent control of separate system components
- Reactive behavior
- Guaranteed response times
- Interaction with special purpose hardware
- Maintenance usually difficult
- Harsh environment
- Constrained resources
- Often cross-development
- Large and complex
- Often have to be extremely dependable

Introduction

> *The more society relinquishes control of its vital functions to computer systems, the more imperative it becomes that those systems do not fail.*
>
> [Burns and Wellings 2001]

- Dependability ("Zuverlässigkeit"):
  - *"The trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers"* [Laprie 1992]
  - *"The metafunctional attributes of a system that relate to the quality of service to its users during an extended interval of time"*
- **Dependability is often the critical aspect of real-time systems!**

Introduction

Aim of this lecture

- Raise awareness of programming language issues (C)
- Introduce concepts relating to dependability
- Provide precise, commonly accepted vocabulary (English and German)

Our baseline here:

- Laprie, J. C. (Ed.), *Dependability: Basic Concepts and Terminology*, Springer, 1992, IFIP (International Federation of Information Processing) Working Group 10.4 on Fault-Tolerant Computing Terms are defined in English, French, German, Italian, and Japanese

Overview

High-Quality Software

Recall:

- Embedded RT applications have particularly high demands on SW *quality*
  - Applications often safety critical
  - Post-deployment modifications often difficult
  - Need **first-time-right** development
- C one of the dominating languages in RT/embedded world
  - SW written directly in C—*or*
  - C programs synthesized from system model

High-Quality Software

- One aim of this class:
  Enable you to write **high-quality** code in C
- Assumes that you already know basics of C
  - Will cover some subtleties of the C language
  - Will also cover proper coding practices and processes
- Apart from C ...
  - ...much of this applies directly to C++, and also Java
  - ...the perspective gained here should be useful for any kind of SW development activity

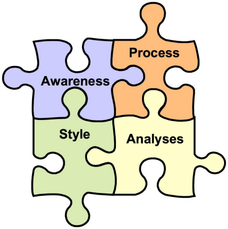
High-Quality C Code

C is like a sharp knife

- Very versatile—but you have to know what you are doing
- Cannot reduce risks arbitrarily without compromising utility

However, can lower risks significantly by

1. Risk awareness
2. Proper development process
3. Conservative coding style
4. Extensive static (compile-time) and dynamic (run-time) analyses



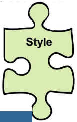
These precautions are all complementary

- None replaces the others

Style Guides

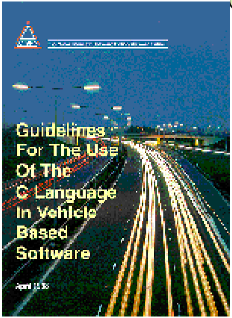
- Larger SW development projects typically enforce some style guide
- Aims:
  - Enhanced readability and maintainability
  - Lower defect rates
- Typically focus on syntactic matters
  - "Variable names should be in lower case"
- However, may also contain semantic rules
  - "The goto statement shall not be used"
  - Effectively define a subset of the implementation language

Style Guides



MISRA "Guidelines for the Use of the C Language in Vehicle Based Software"

- Defined by the Motor Industry Software Reliability Association
- Focuses on embedded/RT applications
- Emphasizes robustness
- Will quote from this throughout class
- `http://www.misra.org.uk`



C Traps and Pitfalls—Lexical Pitfalls

- Individual characters of a program meaningless in isolation
- *Context matters!*

```
p->s = "->"
```

Lexical analyzer

- Part of compiler
- Breaks program into tokens
  - Keywords
  - Variable names
  - Constants
  - etc.
- C permits arbitrary whitespace *between* tokens

Greedy Lexical Analysis

Question: Should "->" be parsed as one or two tokens?

Lexical analysis of C

- Proceeds from left to right
- Always takes *the longest token possible*
- This is the greedy (or maximum munch) rule

```
a = b = 10;
a---b;
```

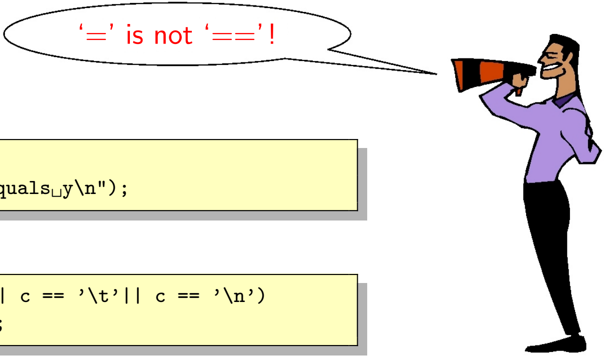
What are the values of `a` *and* `b`*?*

```
y = x/*p /* p points at divisor */;
```

What is the problem?

C Traps and Pitfalls—Lexical Pitfalls

The probably most prominent C trap:




```
if (x = y)
  printf("x␣equals␣y\n");
```

```
if (c = '␣' || c == '\t'|| c == '\n')
  c = getc(f);
```

MISRA C

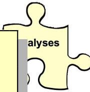
MISRA Rule 35 (required):

Assignment operators shall not be used in expressions which return Boolean values.

Static Code Analysis

- One analysis tool: *The compiler*
- To generate code, compiler must check at least
  - Syntax
  - Types (depending on language)
  - Presence of referenced functions/methods etc.
- Some compilers (esp. in embedded world) restrict themselves to that
- Others (such as gcc) can perform much more detailed analyses—if we ask them to do so!

Static Code Analysis



One aid towards robust code:

- **Harnessing the analyses capabilities of the compiler**

gcc

- Setting "-Wall" flag requests most (but not all) available analyses

```
% cat x-equals-y.c
#include <stdio.h>
int main () {
  int x=1, y=2;
  if (x=y) printf("x␣equals␣y\n");
  return 0;
}

% gcc x-equals-y.c

% gcc -Wall x-equals-y.c
x-equals-y.c:  In function ‘main’:
x-equals-y.c:7:  warning:  suggest
parentheses around assignment
used as truth value
```

C Traps and Pitfalls II—Operators

Scenario 1:

- Consider the following:

  if (flags & FLAG) ...

- Want to make comparison to 0 explicit:

  if (flags & FLAG != 0) ...

- Ooops - the latter actually means

  if (flags & (FLAG != 0)) ...

C Traps and Pitfalls II—Operators

Scenario 2:

- Suppose you want to combine low-order bits of low and high-order bits of hi:

```
r = hi<<4 + low;
```

- However, addition binds more tightly than shifting—the above actually means

```
r = hi << (4 + low);
```

- Correct alternatives:

```
r = (hi<<4) + low;
r = hi<<4 | low;
```

MISRA C

MISRA Rule 47 (advisory):

No dependence should be placed on C's operator precedence rules in expressions.

C Operator Precedence

Unary

Arithmetic

Shift

Relational

Logical

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>() [ ] -> . x++ x--</td><td>left</td></tr>
<tr><td>! ∼ ++x --x + - * & sizeof</td><td>right</td></tr>
<tr><td>(type)</td><td>right</td></tr>
<tr><td>* / %</td><td>left</td></tr>
<tr><td>+ -</td><td>left</td></tr>
<tr><td><< >></td><td>left</td></tr>
<tr><td>< <= > >=</td><td>left</td></tr>
<tr><td>== !=</td><td>left</td></tr>
<tr><td>&</td><td>left</td></tr>
<tr><td>^</td><td>left</td></tr>
<tr><td>|</td><td>left</td></tr>
<tr><td>&&</td><td>left</td></tr>
<tr><td>||</td><td>left</td></tr>
<tr><td>?:</td><td>right</td></tr>
<tr><td>assignments</td><td>right</td></tr>
<tr><td>,</td><td>left</td></tr>
</table>

Java Operator Precedence

Unary

Arithmetic

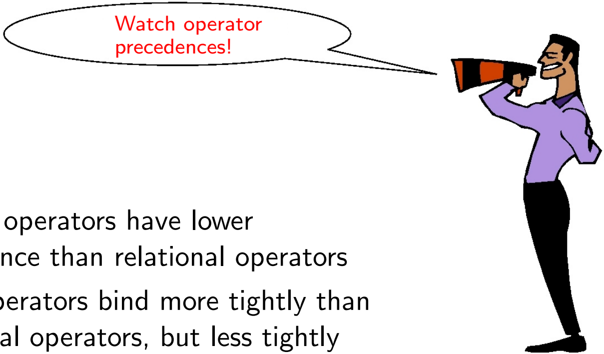
Shift

Relational

Logical

Operator	Associativity
() [ ] . x++ x--	left
! ~ ++x --x + -	right
new (type)	right
* / %	
+ -	left
<< >> >>>	left
< <= > >= instanceof	left
== !=	left
&	left
∧	left
|	left
&&	left
||	left
?:	right
assignments	right

Operator Precedence



Key points:

- Logical operators have lower precedence than relational operators
- Shift operators bind more tightly than relational operators, but less tightly than the arithmetic operators

MISRA C

MISRA Rule 34 (required):

The operands of a logical && or || shall be primary expressions.

Primary expression:

- Single identifiers, constants, parenthesized expressions

C—Order of Evaluation

Precedence

- `a + b * c` interpreted as `a + (b * c)`
  Not: `(a + b) * c`
- Overridden by parentheses

Associativity

- `a + b + c` interpreted as `(a + b) + c`
  Not: `a + (b + c)`
- Overridden by parentheses

Order of evaluation

- `x = (i++, i)` interpreted as `i++; x = i`
  Not: `x = i; i++`
- `if (cnt != 0 && sum/cnt < limit)` does not cause “divide by zero”
- Relevant in the presence of side effects
- Parentheses do not matter!

C—Order of Evaluation

- Undefined results for
  - x = b[i] + i++;
  - x = f(i++, i++);
  - push(pop()—pop());
  - i = ++i + 1;
- Only &&, ||, ?:, and , specify order of evaluation
  - && and || evaluate left op first, right iff necessary
  - a ? b : c evaluates a first, then either b or c
  - a, b evaluates a and discards its value, then b

C—Order of Evaluation


```c
// Example 1
i = 0;
while (i < n)
  y[i] = x[i++];
```

```c
// Example 2
i = 0;
while (i < n)
  y[i++] = x[i];
```

```c
// Example 3
i = 0;
while (i < n) {
  y[i] = x[i];
  i++;
}
```

```c
// Example 4
for (i = 0; i < n; i++)
  y[i] = x[i];
```

MISRA C

MISRA Rule 33 (required):

The right hand operand of a && or || shall not contain side effects.

MISRA C

MISRA Rule 46 (required):

The value of an expression shall be the same under any order of evaluation that the standard permits.

C—Order of Evaluation

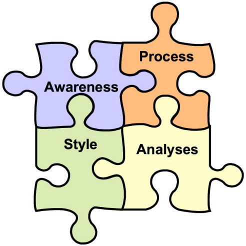
Related issue: number of times a subexpression is evaluated

```
#define MAX(a, b) {((a) > (b)) ? (a) : (b)}
...
z = MAX(i++, j);
```

Arguments to macros should not contain side effects.

C Traps and Pitfalls III

- Language issues addressed so far:
  - Lexical pitfalls
  - Precedence
  - Order of evaluation
- What we did about it so far:
  - Use style guides
  - (Enforce style guides)
  - Ask compiler to generate warnings



C Traps and Pitfalls III

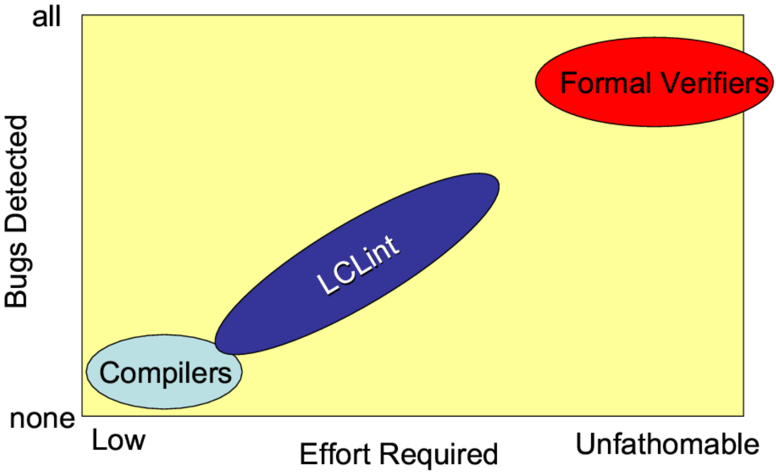
- So, the good news so far:
  - C by now well understood
  - gcc is getting smarter & smarter
  - . . . and programmers worth their salt take advantage of this
- But what about
  - Memory leaks
  - Dereferencing bad pointers
  - Dead code
  - Modifying loop variables
  - Use before definition
  - etc. etc.?
- Good compilers may detect these – but:
  - Typically limited to simple cases
  - Do not know programmer's intent

C Traps and Pitfalls III

Further means that are critical for solid sw development (and that university-educated computer scientists are often ignorant about):

1. Use of additional static checking tools (not only the compiler)
   - The classic: `lint`
   - Can incorporate this into sw development process with minimal effort (e.g., make it part of compilation in Makefile)
2. Clarifying programmer's intent through program annotations
   - "Yes, I really want to modify the loop variable here"
   - Example: Splint (was: lclint) - More on this in the following
3. Use of dynamic checking tools
   - In particular, memory issues often difficult to address statically
   - The classic: `purify`

Introduction to LCLint/Splint



David Evans 2001

Requirements

- No interaction required – as easy to use as a compiler
- Fast checking – as fast as a compiler
- Gradual Learning/Effort Curve
  - Little needed to start
  - Clear payoff relative to user effort

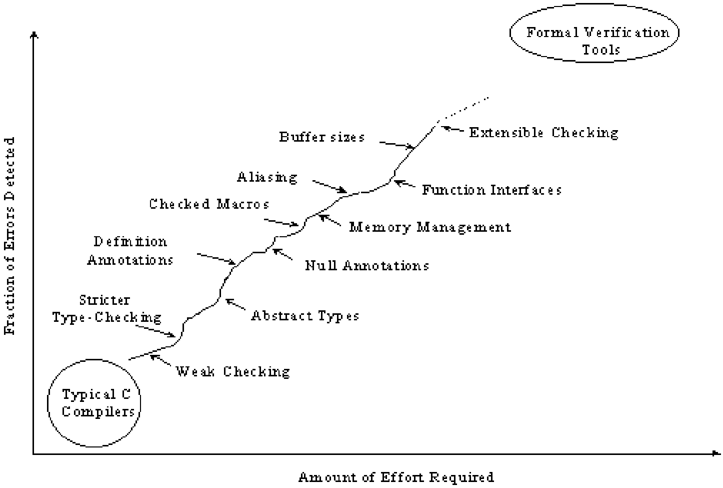
Approach

- Programmers add annotations (formal specifications)
  - Simple and precise
  - Describe programmers intent:
    - Types, memory management, data hiding, aliasing, modification, null-ity, etc.
- LCLint detects inconsistencies between annotations and code
  - Simple (fast!) dataflow analyses

Checking Examples

- Encapsulation – abstract types (representation exposure), global variables, documented modifications
- Memory management – leaks, dead references
- De-referencing null pointers, dangerous aliasing, undefined behavior (order of modifications, etc.)

The Road To Robust C



Summary

- Style guides and a good understanding of the used programming language help to develop reliable, maintainable software
- A first step towards understanding C is to understand the rules for lexical analysis
- One important static analysis tool is the compiler. However, there are other, dedicated static analysis tools (lclint, purify, . . .) that have much more capabilities.

Overview

Basic Terminology

- System ("black box" view):
  - Entity interacting or interfering with other entities (systems)—the environment
- System function:
  - What the system *is intended for*
- System behavior:
  - What the system *does*
- System user:
  - That part of the environment that *interacts* with system

Basic Terminology

- Service of a system:
  - System behavior as perceived by its user(s)
  - An (application dependent) abstraction of the system's behavior
- Timeliness properties are of special interest to dependability
- Real-time function or service:
  - Function or service required to be fulfilled or delivered within finite time intervals dictated by the environment
- Real-time system:
  - System fulfilling at least one RT function or delivering at least one RT service

Basic Terminology

- System ("white box" view):
  - Set of components bound together to interact
- Component:
  - Another system
- Atomic system:
  - Internal structure inexistent/irrelevant

Basic Terminology

- System structure (classical definition):
  - What a system *is*
  - Considers fixed structure
  - Discounts dependability impairments
- System structure (our definition):
  - What *makes it do what it does*
  - Allows for structural changes
- State:
  - Condition of being with respect to a set of circumstances
  - Applies to behavior and structure

Basic Terminology

- Specification:
  - An agreed description of the system's expected function and/or service
  - Also describes admissible conditions (environment, exposure time, performance, etc.)
  - Describes what should be fulfilled/delivered
- For safety/security related systems:
  - Also describes what should *not* happen
  - May in turn lead to specifying additional functions/services that system should fulfill/deliver to reduce likelihood of what should not happen (e. g., user authentication)

Overview

High-Quality C Code

Dependability—Basic Terminology

Attributes of Dependability
- Safety
- Reliability
- Maintainability
- Availability
- Security

Impairments to Dependability

Dependability Requirements

- Dependability (“*Zuverlässigkeit*”) may be viewed to different properties
- Leads to the attributes of dependability:
  - Safety (“*Sicherheit*”)
  - Reliability (“*Funktionsfähigkeit*”, “*Überlebenswahrscheinlichkeit*”)
  - Maintainability (“*Instandhaltbarkeit*”)
  - Availability (“*Verfügbarkeit*”)
  - Security (“*Vertraulichkeit*”, “*Daten-Sicherheit*”)

Safety

Safety: Dependability wrt the *non-occurrence of catastrophic failures*

Typical:

- Need ultra-high reliability
- No single component failure may lead to *critical system failure* (e.g., required by TÜV)

Reliability

Reliability of system: Dependability wrt continuity of service

Given: System operational at time t
Then: Reliability is probability $R(T)$ that system will provide specified service (does not fail) throughout an interval $[t, t + T]$

$$R(T) = e^{-\lambda(T)}$$

- Failure rate:
  Expected number $\lambda(T)$ of system failures for a time interval T
- Mean Time to Failure ($MTTF$):
  If failure rate constant, with $\lambda(T) = \lambda_c T$: $MTTF = \frac{1}{\lambda_c}$
- Ultrahigh reliability: typically $MTTF > 10^9$ hrs

Maintainability

Maintainability: Ease of performing maintenance actions

Given: System with benign failure at time t
Then: Maintainability is probability $M(T)$ that system is repaired within $[t, t + T]$

$$M(T) = 1 - e^{-\mu(T)}$$

- Repair rate:
  Expected number $\mu(T)$ of system repairs for interval T
- Mean Time to Repair ($MTTR$):
  If repair rate *constant*, with $\mu(T) = \mu_c T$: $MTTR = \frac{1}{\mu_c}$
- There is often a conflict between reliability and maintainability
  Example: Hardware modularisation

Availability

Availability:

- Dependability wrt readiness for usage
- Probability A that a system will provide specified service
- Measure of correct service delivery wrt alternation of correct and incorrect service

Mean Time Between Failures ($MTBF$)

$$\boxed{MTBF = MTTF + MTTR}$$

For systems with constant λ and μ:

$$\boxed{A = MTTF/MTBF}$$

Can increase A by increasing $MTTF$ or by decreasing $MTTR$—*or both*

Downtime

Availability corresponds to certain downtime

Availability	Downtime/year	Example Component
90%	>1 month	Unattended PC
99%	$\approx$ 4days	Maintained PC
99,9%	$\approx$ 9 hrs	Cluster
99,99%	$\approx$ 1 hr	Multicomputer
99,999%	$\approx$ 5 mins	Embedded System (PC hw)
99,9999%	$\approx$ 30 secs	ES (special hw)

[Veríssimo and Rodrigues 2001]

Security

- Security:
  - Dependability wrt prevention of unauthorized access and/or handling of information
- Security is a combination of
  - Confidentiality: Prevention of unauthorized disclosure of information
  - Integrity: Prevention of unauthorized amendment/alteration/deletion of information
  - Information availability: the prevention of unauthorized withholding of information
- Traditionally an issue for database/transaction systems
- Increasingly relevant for embedded systems as well (message interception/alteration, property protection)
- Difficult to quantify

Overview

High-Quality C Code

Dependability—Basic Terminology

Attributes of Dependability

Impairments to Dependability

- Fault, error, failure
- Origins of failure
- Another system classification
- Example of a fail-safe system: VOTRICS

Means for Dependability

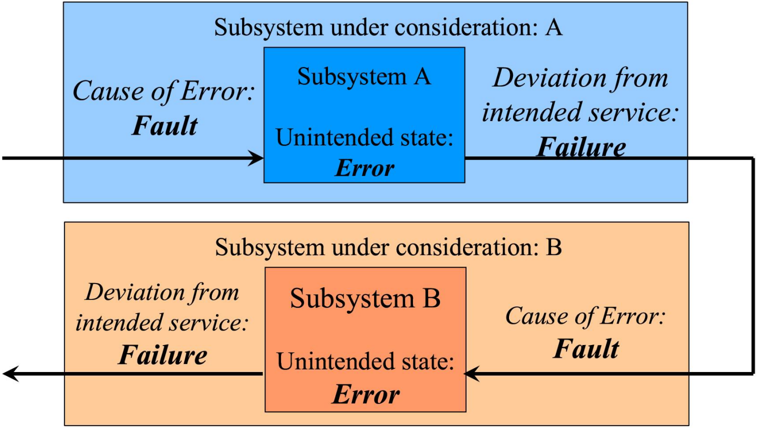
Fault, Error, Failure

- Failure ("Ausfall"):
  - Deviation of actual service (external state) from specification
  - Control surface on wing moves erroneously
  - Airbag does not ignite
- Error ("Fehlzustand"):
  - Unintended (internal) system state liable to lead to subsequent failure
  - Short circuit (excessive current, low voltage)
  - Variable out of range
- Fault ("Fehler"):
  - Adjudged or hypothesized cause of an error
  - Broken isolator, software bug
  - Specification fault

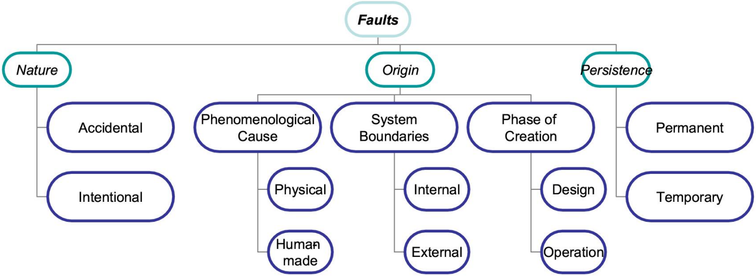
Fault Pathology

- A fault is active when it produces an error
  - Can be internal fault which was dormant and has been activated by computation process
  - Can be external fault
- Errors may be
  - latent: not yet recognized as error
  - detected
- Errors may
  - disappear before detection
  - propagate
- Failures occur when an error passes through system-user interface and affects service
- A component failure results in a fault for
  - The system containing the failed component
  - Other components interacting with the failed component

Sequencing of Fault, Error, Failure



Fault Classification



- Can classify commonly used fault types according to this scheme
  - “intrusions,” “malicious logic,” “physical faults” etc.
- See [Laprie 1992] for more details

Errors

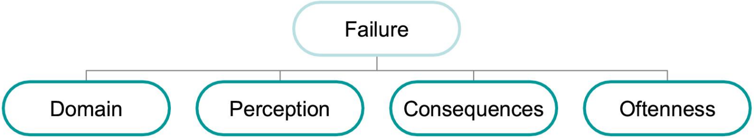
Recall: Error is liable to lead to subsequent failure

Whether or not error actually leads to failure depends on

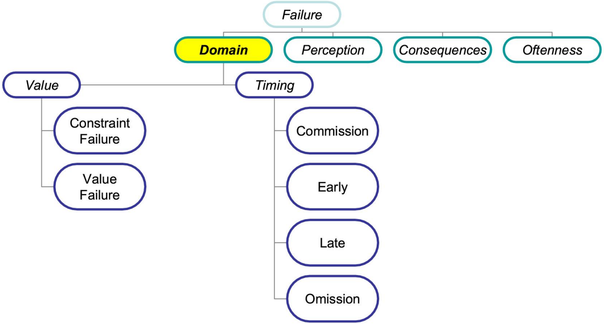
- System composition
  - In particular, (intentional or unintentional) redundancy
- System activity
  - Error may be overwritten before creating damage
- Definition of failure from user's viewpoint
  - "*It's not a bug, it's a feature!*"

Classification of Failures

- A system generally does not always fail in the same way
- Failure modes:
  - The ways a system can fail
  - Can be characterized according to different view points

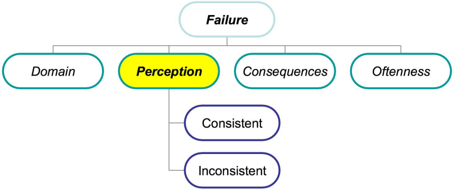


Failure Domain



- Arbitrary failures:
  - Combinations of value and timing domain failures

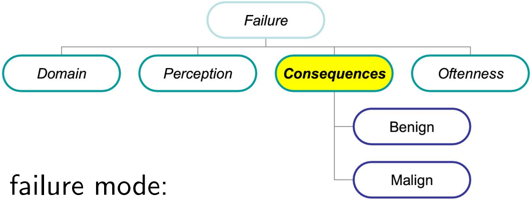
Failure Perception



In system with more than one user:

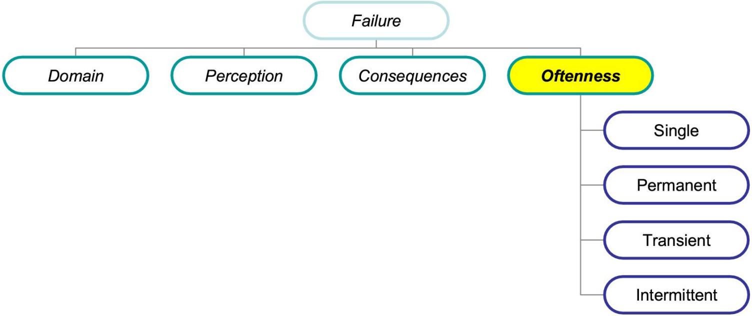
- Consistent failures:
  - Perceptions of the users are the same
- Inconsistent failures:
  - Perceptions are different
  - Also referred to as two-faced failures, malicious failures, or Byzantine failures

Failure Effect



- Benign failure mode:
  - Uncritical failures
- Malign (critical) failure mode:
  - "Cost" of failure exceeds utility of system during normal operation by orders of magnitude
  - Airbags, airtraffic control, nuclear power plants, ...

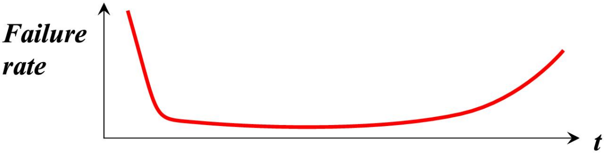
Failure Oftenness



Permanent Failures

A typical VLSI device failure rate develops according to the "bathtub pattern":

- A relatively high failure rate for the first few hundred hours of operation (burn-in)
- After that, stabilization at about 10-100 FIT (= Failures per 10^9 hrs, so 1 FIT corresponds to *MTTF* of about 115 Kyrs)
- At some point, an increased failure rate again (aging)



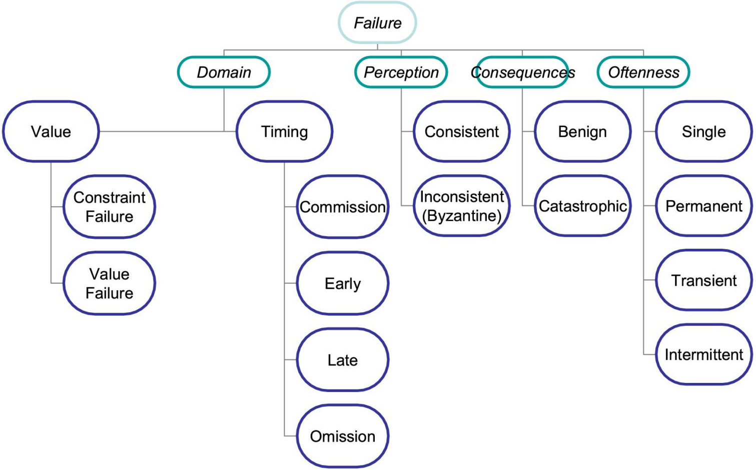
Preventive Maintenance

- Failure rate of a VLSI chip
  - Depends mainly on physical parameters (pins, packaging)
  - Not very sensitive to the number of transistors
- Preventive maintenance
  - Exchange of components before they fail
  - Limits effects of aging
- **If there is no aging, then there is no point in preventive maintenance!**

Transient Failures

- Transient chip failure rate
  - Can be 10–100 000 $\times$ permanent failure rate
  - Depends on physical environment
- Most common causes are
  - Electromagnetic interferences (EMI)
  - Power supply glitches
  - High-energy particles (e.g., α-particles)
- Example from radar monitoring [Gebman et al. 1988]:
  - Malfunctions noticed every 6 flight hrs
  - Maintenance request every 31 hrs
  - Only every 3rd failure could be reproduced!

Failure Classification—Summary



Origins of Failure

- Rule of thumb (JPL data):
  - 1 major fault every 3 pages of requirements
  - 1 major fault every 21 pages of code
- Fault statistics for some NASA space projects:
  - Coding faults: 6% of overall faults (!!!)
  - Function faults: 71% (due to requirements/design problems)
  - Interface faults: 23% (due to poor comm. between teams)
- Observation:
  - **Most severe faults are introduced early but are detected late! (often during system integration)**

Origins of Failure

Results of one study on large information systems (Tandem):

- >40% of failures due to human operator faults
- 25% caused by software faults
- Large contribution by environmental factors
  - Power outages
  - Fires, floods
- Smallest contributor: (random) hardware faults

One of the lessons:

- **Need not only hw fault tolerance, but also sw fault tolerance!**

Another System Classification

- Given: consistent failure perception
- Fail silent: System produces either correct results (both in value and time domains) or no results at all
- Fail crash: Fail-silent system that stops operating after the first failure
- Fail stop: Fail-crash system that makes its failure known to other systems
- Fail (un-)controlled: System that fails in a(n) (un-)controlled manner
- Fail-never: System that always provides correct services in both the timing and value domains
- Fail-safe: System that maintains its integrity in the presence of faults

Example of a Fail-Safe System: VOTRICS

- Train Signalling System developed by Alcatel
- An industrial example of applying design diversity in a safety-critical RT environment
- Objective of train signalling system:
  - Collect data about the state of the tracks in a train station—current position and movements of trains, position of points
  - Set signals and shift points such that trains can move safely through the station according to a given time table

VOTRICS cont.

- VOTRICS is partitioned into two independent subsystems
- First system:
  - Accepts commands from operators
  - Collects data from tracks
  - Calculates intended positions of signals and points
  - Uses a standard programming paradigm
  - Uses a Triple-Mode Redundancy (TMR) architecture to tolerate single HW fault

VOTRICS cont.

- The second system, the "safety bag":
  - Monitors safety of the state of the station
  - Has access to RT data base and intended outputs of 1st system
  - Dynamically evaluates safety predicates derived from the "rule book" of the railway authority
  - Based on expert-system technology
  - Also implemented on TMR HW architecture

VOTRICS cont.

- The two systems exhibit a substantial degree of independence
- Used different specifications as starting point
  - Operational requirements vs. safety rules
- Used different implementation approach
  - Standard programming vs. expert system
- System has been operational in different railway stations for a number of years, *no unsafe state has been detected*

Overview

Means for dependability

Dependability Procurement

- Provide system with ability to deliver service complying with specification
- Fault prevention
  - Prevent faults creeping into a system before it goes operational
- Fault tolerance
  - Provide specified service even in presence of faults

Means for Dependability

Dependability Validation

- Reach confidence in a system
- Attempts to produce systems with well-defined failure modes
- Fault removal
  - Reduce presence (number, seriousness) of faults
- Fault forecasting
  - Estimate present number, future incidence, and consequences of faults

Combination of fault removal and fault prevention also referred to as fault avoidance

Dependencies Between Dependability Means

In general, none of these goals can be achieved perfectly

- All boil down to human activities
- Hence, require combined utilization
- Example: Need for fault-tolerance despite fault avoidance strategies in design process

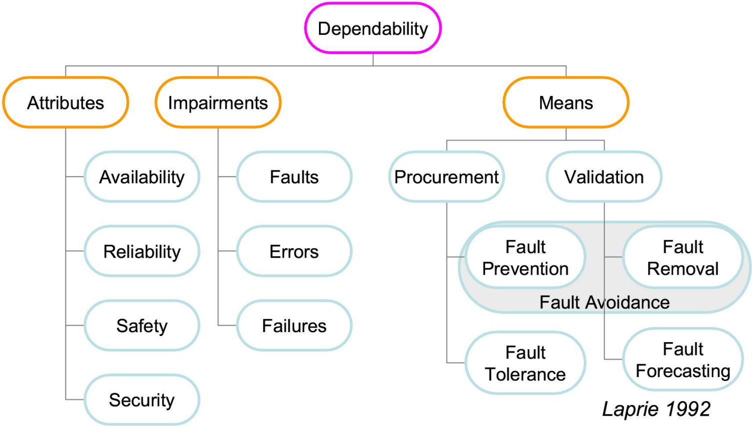
But also: to build dependable systems, tools to develop these systems must be dependable as well

- Example: Certified code generators
- Example: In 1979, an error discovered in program used to design nuclear reactors (supposedly guaranteeing the attainment of earthquake safety standards) resulted in shutting down of 5 nuclear plants [Leveson 1986]

Dependencies Between Dependability Means

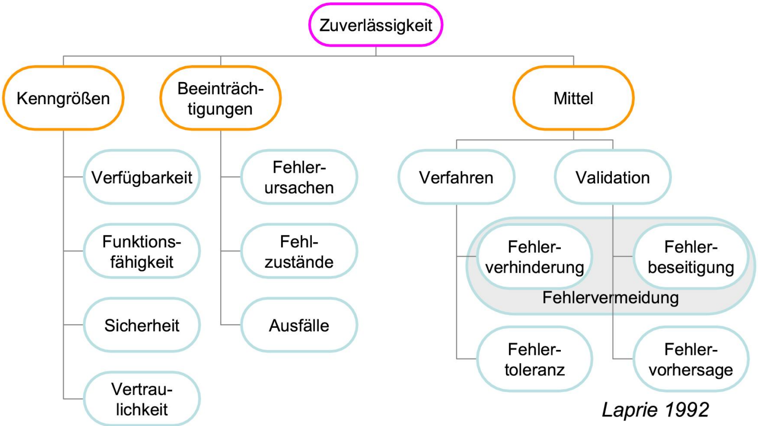
- Interdependencies between fault removal and fault forecasting motivate their combination into dependability validation
- Note, however: fault removal also consists of what is classically referred to as verification
- "V & V" [Boehm 1979]:
  - Verification: Building the system right
  - Validation: Building the right system
- Need validation of the validation
  - Coverage: Measures representativeness of the situations to which the system is submitted during its validation compared to the actual situations it will be confronted with during its operational life

Summary—The Dependability Tree



Laprie 1992

Der Zuverlässigkeitsbaum



To Go Further (1)

Advice on C

- Andrew Koenig, *C Traps and Pitfalls*, Addison Wesley, 1989
- Les Hatton, *Safer C—Development of High-Integrity & Safety-Critical Systems*, McGraw-Hill, 1995 (*currently out of print*)

MISRA C

- `http://www.misra.org.uk`
- Nigel Jones, Introduction to MISRA C, *Embedded Systems Programming*, Jul 1, 2002 (10:07 AM)
- `http://www.embedded.com/columns/technicalinsights/172301672`

To Go Further (2)

- Laprie, J. C. (Ed.), Dependability: Basic Concepts and Terminology, Springer, 1992 (Working Group 10.4 on Fault-Tolerant Computing of the International Federation of Information Processing)
  - Windows-Help on this, courtesy of Jean Claude Laprie (LAAS-CNRS) and Günter Heiner, Jörg Donandt et al. (DaimlerChrysler Berlin), can be found at `http://rtsys.informatik.uni-kiel.de/teaching/open-srcs/LaprieDependabilityTerminology.hlp`
- [Marwedel 2008], Chapter 6.7
- [Veríssimo and Rodrigues 2001], Chapter 6
- [Kopetz 1997], Chapter 6
- [Burns and Wellings 2001], Chapter 5