Reliability

Uwe R. Zimmer – The Australian National University

References for this chapter

[Burns98]
Alan Burns, Brian Dobbing, George Romanski
The Ravenscar Tasking Profile for High Integrity Real-Time Programs
Reliable Software Technologies — Ada-Europe '98, Uppsala, Sweden, June 1998

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Lyu92]
Michael R. Lyu, Algirdas Avizienis
Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming
in: Fault-Tolerant Software Systems: Techniques and Applications, H. Pham (ed.), IEEE Computer Society Press Technology Series, October 1992, pp. 45-54

[Schobbens99]
P.-Y. Schobbens, J.-F. Raskin, T.A. Henzinger, L. Ferier
Axioms for Real-Time Logics
Lecture Notes in Computer Science 1466, Springer-Verlag, 1999, pp. 219-236

all references and links are available on the course page

Based on a set of powerful and diverse tools …

… reconsidering the basic problems of:

- system identification / analysis
- fault prevention
- error detection
- fault tolerance

☞ … building predictable / dependable systems …

… in a real-time domain!

Terminology

Reliability ::=
measure of success with which a system conforms to its specification
or
low failure rate.

Failure ::= deviation of a system from its specification
Error ::= system state which lead to failures
Fault ::= the reason for an error

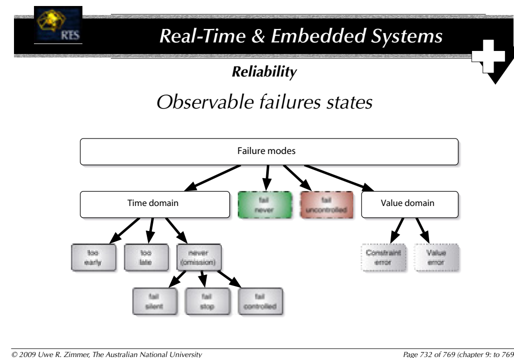
Faults on different levels

- Inconsistent or inadequate specification
  ☞ very frequent source for disastrous faults
- Software design errors
  ☞ very frequent source for disastrous faults
- Component & communication system failures
  ☞ rare and mostly predictable

Faults in the time domain

- Transient faults
  ☞ many communication system failures, electric interference, etc.
- Intermittent faults
  ☞ transient errors which occur more than once (e.g. overheating effects)
- Permanent faults
  ☞ stay in the system until they are repaired by some means

Observable failures states



Achieving reliability

System identification

Investigate:

- static applications specifications
- physical sensors and converters constraints
- constraints of the employed controller network
- constraints of the underlying run-time system
- dynamic application specifications (requested real-time behaviour)

☞ Understanding all critical real-time requirements and issues

Fault avoidance

Fault avoidance at hardware-level:

- use reliable hardware components — consider the environmental demands!
- use an adequate (hardware) system design — shock, humidity, interference, …
- ensure proper assembly and encapsulation — weak connectors, bad connectors, …

Fault avoidance at software design level:

- strict system specifications (employ format methods if applicable)
- use proven software-engineering and design methodologies
- employ languages and run-time environments with reasonable support for the requirements.

Fault removal

Find and remove errors from the previous stage.

☞ Team programming methods like extreme programming or rigorous testing may help here.

but …

- no re-evaluation method indicates the absence of faults
  (even formal methods cannot identify specification faults)

… and specifically for real-time and embedded systems …:

- often: tests cannot be performed under realistic conditions … especially exceptional conditions
- most simulation environments have a severe impact on real-time systems
- the test space for real-time system is significantly larger than for non-real-time systems

Fault prevention

(avoidance & removal)

Regardless of the rigor of fault prevention methods:

the real-time system might still fail

This is specifically critical for all non-monitored systems:

- systems which are (temporary) inaccessible
- un-manned vehicles which operate autonomously by default
- systems in remote / dangerous environments

Instead (or in addition to fault prevention): enabling a 'safe landing': ☞ Fault tolerance

Fault tolerance

- Full fault tolerance
  the system continues to operate in the presence of 'foreseeable' error conditions without any significant failures — also this might induct a reduced operation period.
- Graceful degradation (fail soft)
  the system continues to operate in the presence of 'foreseeable' error conditions, accepting a partial loss of functionality or performance.
- Fail safe
  the system halts and maintains its integrity

☞ Full fault tolerance is not maintainable for an infinite operation time!

☞ Graceful degradation might have multiple levels of reduced functionality.

Hardware redundancy

☞ adding extra hardware resources:

- for the **detection** of failures and the localization of faults
- for the **handling** of exceptional situations and error-recovery.
- as a functional duplication or multiplication of complete (sub-)systems in order to **hot-swap** or **select** the operational one in case of a failure in one part of the (sub-)system.

- Fault-detection and recovery hardware includes:
  watch-dog timers, limit switches, additional physical sensors, transient-recording-systems (emergency system dump), overload-backup-systems, or even in-circuit emulators.
- Triple Modular Redundancy (TMR) or N-Modular Redundancy (NMR) assumes:
  functionally identical components which are either:
  - static parts of the system and connected via a voting/masking/comparing system
  - or in case of a detected error-condition: dynamic parts which are swapped in.

Hardware redundancy

☞ any hardware redundancy adds to the overall system complexity!

In case of TMR or NMR:

☞ the assumption that an error occurs in one part of the system only requires that either:

- the fault is based on a physical phenomenon, which applies only locally
- or the structure of the functionally identical systems is sufficiently different

For some high-risk systems this approach is applied in forms of redundant sub-systems with:

- the same specification
- different computer systems (CPUs, buses, memory systems, drives)
- different operating systems
- different real-time languages and development environments (N-Version programming)
- and by restricting the communication between the different developer teams

not too surprisingly, the outputs from the different systems are slightly different …

Triple Modular Redundancy

(example)

3 identical primary flight computers distributed in the Boeing 777, each consisting of:

- 3 processors: AMD 29050, Motorola 68040, INTEL 80486 (called 'lanes')
- independent power-sources and inertia measurements
- code build by 3 different Ada compilers
- *the same Ada source code* ('the specification'): around 3 million lines of code, but different monitor functions

Targeted failure probability: $< 10^{-10}/h$ (e.g. UK Seizewell B nuclear reactor (emerg.): $< 10^{-3}/h$)
No single fault on board the 777 should occur without failure identification.
No single fault on board the 777 should cause more than the loss of one primary flight computer.

☞ Sophisticated synchronization and communication systems.

(not a single fatal event — information from November 2001)

N-version programming

Impacts to software diversity:

	Development teams	Languages	Tools	Algorithms	Methodologies
Specification	high	high	lowest	highest	low
Design	highest	low	low	highest	high
Coding	highest	highest	low	high	high
Testing	low	lowest	high	low	highest

(■: highest – ■: high – ■: low – ■: lowest) (source: [Lyu92])

"The six-language project"

Joint project between the UCLA (Dependable computing and fault-tolerance systems) and the Honeywell Commercial Flight Systems Division (1992)

- The specifications (about a flight controller) were original system description documents (SDD) by Honeywell enhanced by additional cross-checking points and included some enforced diversity elements (a 64-page document).
- The development teams were isolated and any technical discussions were strictly prohibited
- All communication and documentation is requested to follow predefined protocols (written form) defined and handled by a coordinating team.
- Specified tests were performed by the coordinating team before a version was accepted for integration.
- The *N*-Version paradigm was applied to all stages of the development cycle.

"The six-language project"

(source: [Lyu92])

Language	Sources (l.o.c.)	Test runs	Errors	Failure-rate
Ada	2256	5127400	0	0
'C'	1531	— " —	568	1.108×10^{-4}
Modula-2	1562	— " —	0	0
Pascal	2331	— " —	0	0
Prolog	2228	— " —	680	1.326×10^{-4}
T (close to Lisp)	1568	— " —	680	1.326×10^{-4}
Average	1913	— " —	321	0.627×10^{-4}

"The six-language project"

(source: [Lyu92])

Failure category	average single version failure probabilities (5127400 cases)	average 3-version failure probabilities (102548000 cases)	average 5-version failure probabilities (30764400 cases)
no errors	0.99993733	.9998409	.9997807
single error	6.27×10^{-5}	13.05×10^{-5}	19.15×10^{-5}
two distinct errors		00.20×10^{-5}	00.23×10^{-5}
two coincident errors		02.65×10^{-5}	02.21×10^{-5}
three errors			00.34×10^{-5}

"The six-language project"

(source: [Lyu92])

☞ The resulting 3-version and 5-version systems displayed lower failure rates than a 'golden master' reference implementation by Honeywell.

☞ Coincident errors involving more than two versions were never observed.

☞ a total of 93 faults were detected.

☞ control problems are specifically suitable for n-version programming, since the error-detection and synchronization algorithms are relatively simple.

in general: diverting results do not necessarily imply any faults.

N-version programming – Voting issues

- **Integer arithmetic:**
  - Integer (or any discrete sub-type) -based results will be identical ➩ ✔
- **Real arithmetic:**
  - Real-valued results will usually be different ➩ Comparisons need to consider tolerances.
  - If the process is not fully continuous (thresholds, quantizations, bifurcations)
    ➩ Comparisons need to re-model the whole process in order to evaluate similarities
    ➩ Independence ✗ re-specify the system
- **Multiple solutions:**
  - The solution space itself allows for multiple correct, but different solutions
    ➩ ✗ re-specify the system

N-version programming

some issues:

- **Specification:**
  Assuming that a good part of software faults stem from wrong or incomplete specifications
  ☞ *N*-version programming will not help in this case
- **Diversity assumption:**
  Diversity can be enforced and supported in some areas (demonstrated by examples), while co-incident error conditions can be observed in other application domains (also documented by case-studies). The rigorous identification of adequate domains for *N*-version programming is currently part of active research.
- **Project costs:**
  Since the development costs are increasing by a factor of *N* plus coordination costs, it needs to be considered carefully whether a single version developed with the same effort shows perhaps a similar level of reliability.

Dynamic redundancy

Four constituent phases (Anderson and Lee, '90):

1. **Error detection**
   Detection of a precise error state is essential.
2. **Damage confinement and assessment**
   Diagnosis of the damage, which occurred between the fault and the detected error state.
3. **Error recovery**
   Sequence of operations leading from the detected error state to an operational state.
4. **Fault treatment**
   In order to prevent the same error state again, the fault itself might/should be eliminated.

Dynamic redundancy — Error detection

- Error states from the environment
  - **Hardware** … CPU, controllers, communication systems, …
  - **Run-time environment**
- Error states stemming from checks without the application processes
  - **Replication** – employ N-version programming to detect error states
  - **Timing** – watchdog timers and overrun detectors
  - **Reversal** – apply the reverse function and compare $x \Leftrightarrow f^{-1}(f(x))$
  - **Coding** – detect corrupted data via redundant information (CRC-checks, …)
  - **Reasonableness** – check assertions (e.g. in Eiffel)
  - **Structural** – check structural integrity (e.g. lists, file-systems)
  - **Continuity** – assuming a limited difference between consecutive controller values.
  - …

Dynamic redundancy — Damage diagnosis

Confinement:

☞ How to avoid the transfer of fault-effects between system parts?

- Modular decomposition
- Atomic actions
- 'Firewalls'

Assessment:

☞ resulting from the location of the detected error state and the possible paths though the system which are all leading to this error state.

☞ a fine-granular system structure (error-confinement) limits the length of these possible paths.

☞ **a very well structured system is the cornerstone of damage diagnosis.**

Dynamic redundancy — Error recovery

Backward error recovery:

- set checkpoints and safe the system state with each passing of a checkpoint. how can system-wide consistent checkpoints be ensured?
- if a error state is detected: set back to the last consistent checkpoint.

☞ applicable even if the fault itself can not be identified.

☞ not applicable at all, if the system contains non-reversible or -resetable components (time, …)

Forward error recovery:

- method of choice for most time critical parts of real-time and embedded systems.

☞ highly application dependent.

☞ may involve complex mode and priority changes (deadlines might be still relevant).

Dynamic redundancy — Fault treatment

- Localization of a hardware fault is usually easier and more precise than of a software fault.
- On-line fault treatment might be tricky and is usually limited to (hot) exchanges of complete modules (software as well as hardware).
- Granularity is usually finer than in static redundant systems.
- Exchange of faulty components is nevertheless usually an expensive and complex operation.

☞ the number of substitutable sub-systems in a dynamic redundant system is still limited.

(many systems will assume transient faults, log the event and continue operations …)

Safety and Dependability

- **Safety:** *freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm* (Leveson, '86)

☞ are there *any* safe and functional systems beyond a certain complexity? … aeroplanes? cars?

- **Dependability:**
  - Availability — ready to use
  - Reliability — absence of failures
  - Safety — absence of fatal failures
  - Confidentiality — absence of unauthorized disclosures
  - Integrity — no data corruptions
  - Maintainablity — accessibility to changes and improvements

… more reliability in the design process:

Restrict, Formalise, …?

Restrict:

- limit the tools and environments to 'safer' operations
- e.g. Esterel, High-Integrity Pearl, Ada95 Ravenscar profile, …

Formalise:

- UML ('the object oriented approach')
- Temporal logic, Real-Time Logic (RTL) as an extension of predicate logic
- classical real-time design methods: MASCOT, JSD, MOON, HOOD, HRT-HOOD, CODARTS, …

Ada95 Ravenscar profile (Burns, Dobbing, Romanski '98)

- ***Task type and object declarations at the library level***
  — ☞ no hierarchy of tasks, and hence no exit protocols needed from blocks and sub-programs.
- ***No dynamic allocation or unchecked de-allocation of protected and task objects***
  — removes the need for dynamic objects.
- ***Tasks are assumed to be non-terminating***
  — this is primarily because task termination is generally considered to be an error for a real-time program which is long-running and defines all of its tasks at start-up.
- ***Library level Protected objects with no entries***
  — these provide atomic updates to shared data and can be implemented simply.
- ***Library level Protected objects with a single entry***
  — used for invocation signalling; but removes the overheads of a complicated exit protocol.

Ada95 Ravenscar profile (Burns, Dobbing, Romanski '98)

- ***Barrier consisting of a single boolean variable***
  — no side effects are possible and exit protocol becomes simple.
- ***Only a single task may queue on an entry*** — hence no queue required; this is a static property that can easily be verified, or it can lead to a bounded error at runtime.
- ***No requeue*** — leads to complicated protocols, significant overheads and is difficult to analyse (both functionally and temporally).
- ***No Abort or ATC*** — these features leads to the greatest overhead in the run-time system due to the need to protect data structures against asynchronous task actions.
- ***No use of the select statement*** — non-deterministic behaviour is difficult to analyse, moreover the existence of protected objects has diminished the importance of the select statement to the tasking model.
- ***No use of task entries*** — not necessary to program systems that can be analysed; it follows that there is no need for the accept statement.

Ada95 Ravenscar profile (Burns, Dobbing, Romanski '98)

- ***"Delay until" statement but no "delay" statement***
  — the absolute form of delay is the correct one to use for constructing periodic tasks.
- ***"Real-Time" package*** — to gain access to the real-time clock.
- ***No Calendar package*** — "Real-Time" package is sufficient.
- ***Atomic and Volatile pragmas*** — needed to enforce the correct use of shared data.
- ***Count attribute (but not within entry barriers)***
  — can be useful for some algorithms and has low overhead.
- ***Ada.Task_Identification*** — can be useful for some algorithms and has low overhead, available in reduced form (no Abort_Task or task attribute functions Callable or Terminated).
- ***Task discriminants*** — can be useful for some algorithms and has low overhead.
- ***No user-defined task attributes***
  — introduces a dynamic feature into the run-time that has complexity and overhead.

© 2009 Uwe R. Zimmer, The Australian National University

Ada95 Ravenscar profile (Burns, Dobbing, Romanski '98)

- ***No use of dynamic priorities*** — ensures that the priority assigned at task creation is unchanged during the task's execution, except when the task is executing a protected operation.
- ***Protected procedures as interrupt handlers*** — required if interrupts are to be handled.

☞ commercially available and employed in hard-real-time systems

© 2009 Uwe R. Zimmer, The Australian National University

Temporal logic

- Extending predicate logic
- Adding the concepts of ordering for events and states
- Suitable for event driven system, reactive systems ☞ Esterel

© 2009 Uwe R. Zimmer, The Australian National University

Temporal logic

- Assertions on sequences and orders of states
  ☞ employ predicate logic & a set of new operators:

$\Box A$: A is *true* for all future states
$\Diamond A$: A is eventually *true*
$\bigcirc A$: A is *true* for the following state

e.g. $\Box(Collission_Warning \Rightarrow \Diamond Collission_Avoidance)$
or: $\Box(Collission_Warning \Rightarrow \bigcirc Collission_Avoidance)$

assuming that there is a sequence of distinguishable states (or 'time') S.

© 2009 Uwe R. Zimmer, The Australian National University

Temporal logic

- Another temporal operator:

  $A \mu B$: A holds until the first occurrence of B, which will occur eventually.

e.g.
$\Box((Tasks_Waiting\ \mu\ Entry_Closed) \wedge (\neg Tasks_Waiting\ \mu\ Entry_Open))$

☞ Temporal logic expresses the order of events only
and has means to express temporal scopes, deadlines, …

© 2009 Uwe R. Zimmer, The Australian National University

Real-Time Logic

- Assertions on real-time events:
  ☞ employ predicate logic & a occurrence function:

  $@(E, i)$: denotes the *time* of the i-th occurrence of event (-class) E

☞ the event (-class) E is strictly ordered by instance (i) and time ($@$).

☞ all events of kind (class) E can be distinguished.

☞ instance order $\Rightarrow$ order in time.

© 2009 Uwe R. Zimmer, The Australian National University

Real-Time Logic

- Occurrence times of predicates:

  $\uparrow A$: denotes the *time* when A changes from *false* to *true*.
  $\downarrow A$: denotes the *time* when A changes from *true* to *false*.

Examples:

$\forall i \exists j\ ((@(E, i) \leq @(\uparrow A, j)) \wedge (@(\downarrow A, j) \leq @(E, i) + d) \wedge (@(\downarrow A, j\text{-}1) \leq @(E, i))$
$\quad \wedge \forall\ i\ (@(E, i{+}1) \geq @(E, i){+}p))$

$\forall i,j\ ((@(\downarrow A, i) < @(\uparrow B, j)) \vee (@(\downarrow B, j) < @(\uparrow A, i)))$

Interpretations:
does $\forall i\ @(E, i)$ indicate all possible, all defined, or all observed instances of E?

© 2009 Uwe R. Zimmer, The Australian National University

Linear Temporal Logic of Real Numbers (LTR)

$$\phi ::= p \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 U \phi_2 \mid \phi_1 S \phi_2$$

where

$(\tau, t) \models p$ iff $p \in \tau(t)$
$(\tau, t) \models \phi_1 \vee \phi_2$ iff $(\tau, t) \models \phi_1$ or $(\tau, t) \models \phi_2$
$(\tau, t) \models \neg\phi$ iff $(\tau, t) \not\models \phi$
$(\tau, t) \models \phi_1 U \phi_2$ iff $\exists t' > t \wedge t' \models \phi_2$ and $\forall t'' \in (t, t'), t'' \models \phi_1 \vee \phi_2$
$(\tau, t) \models \phi_1 S \phi_2$ iff $\exists t' < t \wedge t' \models \phi_2$ and $\forall t'' \in (t', t), t'' \models \phi_1 \vee \phi_2$

ϕ is satisfiable iff $\exists(\tau, t) \models \phi$ — ϕ is valid iff $\forall(\tau, t) \models \phi$

© 2009 Uwe R. Zimmer, The Australian National University

Event-Clock Temporal Logic

$$\phi ::= p \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 U \phi_2 \mid \phi_1 S \phi_2 \mid \lhd_I \phi \mid \rhd_I \phi \mid$$

where

$(\tau, t) \models p$ iff $p \in \tau(t)$
$(\tau, t) \models \phi_1 \vee \phi_2$ iff $(\tau, t) \models \phi_1$ or $(\tau, t) \models \phi_2$
$(\tau, t) \models \neg\phi$ iff $(\tau, t) \not\models \phi$
$(\tau, t) \models \phi_1 U \phi_2$ iff $\exists t' > t \wedge t' \models \phi_2$ and $\forall t'' \in (t, t'), t'' \models \phi_1 \vee \phi_2$
$(\tau, t) \models \phi_1 S \phi_2$ iff $\exists t' < t \wedge t' \models \phi_2$ and $\forall t'' \in (t', t), t'' \models \phi_1 \vee \phi_2$
$(\tau, t) \models \lhd_I \phi$ iff $\exists t' < t \wedge t' \in t - I \wedge t' \models \phi$ and $\forall t'' \in (t - I, t), t'' \not\models \phi$
$(\tau, t) \models \rhd_I \phi$ iff $\exists t' > t \wedge t' \in t + I \wedge t' \models \phi$ and $\forall t'' \in (t, t + I), t'' \not\models \phi$

© 2009 Uwe R. Zimmer, The Australian National University

Metric-Interval Temporal Logic

$$\phi ::= p \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 \hat{U}_I \phi_2 \mid \phi_1 \hat{S}_I \phi_2$$

where

$(\tau, t) \models p$ iff $p \in \tau(t)$
$(\tau, t) \models \phi_1 \vee \phi_2$ iff $(\tau, t) \models \phi_1$ or $(\tau, t) \models \phi_2$
$(\tau, t) \models \neg\phi$ iff $(\tau, t) \not\models \phi$
$(\tau, t) \models \phi_1 \hat{U}_I \phi_2$ iff $\exists t' \in (t, t + I) \wedge t' \models \phi_2$ and $\forall t'' \in (t, t'), t'' \models \phi_1$
$(\tau, t) \models \phi_1 \hat{S}_I \phi_2$ iff $\exists t' \in (t - I, t) \wedge t' \models \phi_2$ and $\forall t'' \in (t', t), t'' \models \phi_1$

© 2009 Uwe R. Zimmer, The Australian National University

Reliability

- **Terminology**
  - Faults, Errors, Failures – Reliability
- **Faults**
  - Fault avoidance, removal, prevention ☞ Fault tolerance
- **Redundancy**
  - Static (TMR, NMR) and dynamic redundancy
  - N-version programming, and dynamic redundancy in software design
- **Reduce & Formalise**
  - Ada95 Ravenscar profile
  - Real-time Logic

© 2009 Uwe R. Zimmer, The Australian National University