Reliability

Reliability, failure & tolerance

Terminology of failure \\
'Failing terminology'?

Reliability ::= measure of success \\
with which a system conforms to its specification.

Failure ::= a deviation of a system from its specification.

Error ::= the system state which leads to a failure.

Fault ::= the reason for an error.

Faults during different phases of design

- Inconsistent or inadequate specifications  
  - frequent source for disastrous faults

- Program errors  
  - frequent source for disastrous faults

- Component & communication system failures  
  - rare and mostly predictable

Faults in the time domain

- Transient faults  
  - Single 'glitches', interference, ... very hard to handle

- Intermittent faults  
  - Faults of a certain regularity ... require careful analysis

- Permanent faults  
  - Faults which stay ... the easiest to find

Achieving reliability

References for this chapter

[Burns98] Alan Burns, Brian Dobbing, George Romanski

[Filliatre2013] J.-C. Filliatre
Axioms for Real-Time Logic
J.-C. Filliatre

Integrity Real-Time Programs
Reliable Software Technologies,
S. T. Taft
Sparkel Reference Manual
Draft 0.6, August 2013

 {[Burns98] [Schobbens99] [Filliatre2013] [Lyu92] [Tara99]}

... consider the basic problems of:

- System identification / analysis
- Fault prevention
- Error detection
- Fault tolerance

... and determine how to build:

- Predictable / dependable systems ... in the real-time domain!

Terminology of failure or 'Failing terminology'?
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).
- To understand all critical real-time requirements and issues.

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 connections, bad PCBs, ...

Fault avoidance at software-design level:
- Verify consistency of specifications (employ formal methods where applicable).
- Tests can often not be performed under realistic conditions — especially exceptional conditions.
- Simulation environments frequently have a severe impact on real-time behaviours.
- No re-evaluation method guarantees the total absence of faults.

Fault prevention, avoidance, removal, …

Hardware redundancy

Regardless of the rigor of fault prevention methods:

The actual real-time system might still fail!

This is specifically critical for unmanned systems:
- Systems which are (temporary) inaccessible.
- Unmanned vehicles which operate semi-autonomously by default.
- Systems in remote/hazardous environments.

Fault tolerance

Full fault tolerance:
- the system continues to operate in the presence of foreseeable error conditions, without any significant loss of functionality or performance — even though this might reduce the achievable total operation time.

Graceful degradation (fail soft):
- the system continues to operate in the presence of unforeseeable error conditions, while 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.

 fault tolerance

N-Modular Redundancy (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.
- 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 specifications.
- Different computer systems (CPUs, buses, memory systems, drivers).
- Different operating systems.
- Different real-time languages and development environments (N-Version programming).
- … and by restricting the communication between the different developer teams.

Fault-tolerant systems which are (temporary) inaccessible.

Fault-tolerance fails if:
- A fault occurs in the system’s fault detection and recovery hardware.
- A fault occurs in one of the redundant sub-systems.
- A fault occurs in the switching logic.
- A fault occurs in the communication logic.

Triple Modular Redundancy (TMR) or N-Modular Redundancy (NMR)

Fault tolerance

Hardware redundancy

For some high-risk systems, this approach is applied in forms of redundant sub-systems with:
- The same specifications.
- Different computer systems (CPUs, buses, memory systems, drivers).
- Different operating systems.
- Different real-time languages and development environments (N-Version programming).
- … and by restricting the communication between the different developer teams.

Fault-tolerant systems which are (temporary) inaccessible.

Fault-tolerance fails if:
- A fault occurs in the system’s fault detection and recovery hardware.
- A fault occurs in one of the redundant sub-systems.
- A fault occurs in the switching logic.
- A fault occurs in the communication logic.

Triple Modular Redundancy (TMR) or N-Modular Redundancy (NMR)

Assumes functionally identical components which are either:
- Static parts of the system and connected via writing/reading systems.
- or in case of a detected error, a critical part which is swapped in.

Any hardware redundancy adds to the overall system complexity!
Reliability

Fault tolerance

“The six-language project”
Joint project between the UCLA (Dependable computing and fault-tolerance systems) and the Honeywell Commercial Flight System Division (1992):

- The specifications (about a flight control) were original system description documents (OSD) by Honeywell enhanced by additional cross-checking points and included some enhanced diversity elements as language documents.
- The development teams were isolated and any technical discussions were strictly prohibited.
- All communication and documentation is required 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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Loc.</th>
<th>Test num</th>
<th>Errors</th>
<th>Failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>2256</td>
<td>5127400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1533</td>
<td>-</td>
<td>568</td>
<td>1.18E-4</td>
</tr>
<tr>
<td>Modula-2</td>
<td>1562</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pascal</td>
<td>2533</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>2228</td>
<td>-</td>
<td>666</td>
<td>1.32E-4</td>
</tr>
<tr>
<td>T (close to Lisp)</td>
<td>1966</td>
<td>-</td>
<td>666</td>
<td>1.32E-4</td>
</tr>
<tr>
<td>Average</td>
<td>1913</td>
<td>-</td>
<td>321</td>
<td>6.27E-3</td>
</tr>
</tbody>
</table>

Failure category:

- Single version
- 3-version
- 5-version

N-Version paradigm was applied to all stages of the development cycle.

N-version programming – Voting issues

Integer arithmetic:

- Integral (or any discrete-type) - based results will be identical.

Real arithmetic:

- Real-valued results will usually be different.

Comparisons need to consider tolerances.

Multiple solutions:

- The solution space itself allows for multiple correct, but structurally different solutions.

N-version programming – Other 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.

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 an 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
   - Correction of operations leading from the detected error state to an operational state.

4. Fault treatment
   - In order to prevent the same error state to occur, the fault itself must be eliminated.

Error states from the environment:

- Hardware - CPU, controllers, communication systems, ...
- Software - check contracts (e.g. in Ada, Spark).
- Structural - check structural integrity (e.g. lists, file-systems).

Error detection:

- Error states stemming from checks within the application processes.
  - Replication - employ N-version programming to detect error states.
  - Timing - check timing, and over run detectors.
  - Recovery - apply the reverse function and compare.

Coding - detect corrupted data via redundant information (CRC-checks, ...).

Confinement:

- Assessing - assuming a limited difference between consecutive controller values.

Confinement and assessment
Reliability

Fault tolerance

Dynamic redundancy

Safety and Dependability

Error recovery

Fault treatment

Freedom

Backward: injury, occupational illness, damage to (or loss of) equipment

Adjust, avoid, correct, or exchange:

If an error state is detected: set back to the last consistent checkpoint.

Localization of a hardware fault is usually easier and more precise than of a software fault.

Online 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 usually an expensive and complex operation.

The number of substitutable subsystems in a dynamic redundant system is limited.

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

Not applicable at all, if the system contains non-reversible components (time, ...)

Forward: — ready to use

— absence of failures

— no data corruptions

— accessibility to changes and improvements

Restrict & Formalize

Further refinements in the design tool chain:

Restrict, Formalize, ...?

Restrict:

- Limit the tools and environments to ‘safe’ operations.

- e.g. Stencil, High Integrity Pearl, Ada Ravenscar profile, Ada DO178 profile runtime...

Formalize:

- Temporal logic, Real-Time Logic (RTL) as an extension of predicate logic.

- Classical real-time design and certification methods: MAGIC, JSI, MOOS, HODD, HIGHWOOD COinati, DO178, ...

Expand:

- Expand a language by means of provable predicates SPARK, Sparkel, Ppassi, Why,...

Ada Zero Footprint profile

No tasking

No scheduling, no task switches, ...

No dynamic allocation

No need for dynamic heap management.

No dynamic dispatching

All bindings happen at compile time.

No exception propagation

No local exception handlers are still permitted.

Packed array:

No packed component of component sizes of powers of two

Reduced code complexity.

Ada Ravenscar profile

Task type and object declarations at the library level:

No hierarchy of tasks, and hence no exit protocols needed from blocks and subprograms.

No dynamic allocation or unchecked de-allocation of protected and task objects

No need for dynamic objects.

Tasks are assumed to be non-dominating

Tasks are treated as non-dominated.

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

These provide atomic updates to shared data and can be implemented simply.

Barrier consisting of a single boolean variable

No side effects are possible and exit protocol becomes simple.

Ada Ravenscar profile

No use of task entries

No need for program systems that can be analyzed; it follows that there is no need for the accept statement.

“Only until” statement but no “delay” statement

No need to embed the time overhead in the delay statement.

“Real-Time” package only

No use of task entries

No need to embed the time overhead in the delay statement.
Ada Ravenscar profile

- Ada Task Identification
- Can be useful for some algorithms and has low overhead in reduced form (no Abort Task or task attribute functions "Callable" or "Terminated")
- Task discrimination
- Can be useful for some algorithms and has low overhead
- No user-defined task attributes
- Extends a dynamic feature into the run-time that has complexity and overhead.
- 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
- No user-defined task attributes
- Extends predicate logic.

Ada Certification profiles

Profiles tailored to specific certification processes, e.g.

- DO-178B, DO-178C
- Managed by the Radio Technical Commission for Aeronautics (RTCA).
- Federal Aviation Administration (FAA)
- European Aviation Safety Agency (EASA)

Address e.g:
- Model driven design, verification, formal methods
- Components can be certified to different levels of assurance depending on fault impact levels:
  - No safety effect, minor, major, hazardous or catastrophic

Temporal logic

- Extending predicate logic:
- Adding a concept of ordering for events and states.
- Suitable for event-driven systems, reactive systems

Real-Time Logic

- Occurrence times of predicates
- A denotes the time when A changes from false to true.
- A denotes the time when A changes from true to false.

Linear Temporal Logic of Real Numbers (LTR)

\[ \phi := p \lor (\phi_1 \land \phi_2) \lor (\phi_1 \lor \phi_2) \lor (\phi_1 \leftrightarrow \phi_2) \lor \neg \phi \lor \exists x \geq 0 \phi \lor \forall x \geq 0 \phi \lor (\exists x \geq 0 \phi) \lor (\forall x \geq 0 \phi) \]

Interpretations:
- All possible, all defined, or all observed instances of E!
**Formalize**

**Metric-Interval Temporal Logic**

\[ \psi = \rho (\psi \lor \psi_1) \downarrow \psi_2 \]

where:

\( (r,t) > \rho \) \iff \( \rho \in [r,t) \)

\( (r,t) > \psi \lor \psi_1 \) \iff \( (r,t) > \psi \) or \( (r,t) > \psi_1 \)

\( (r,t) > \psi \) \iff \( \exists \rho \in (r,t) \land (r,t) > \psi \)

\( (r,t) > \psi_1 \) \iff \( \exists r' \in [r,t) \land \exists C \in [r,t) \land (r,t) > \psi_1 \)

\[ \phi \text{ is satisfiable \iff } (r,t) > \phi \]

\[ \phi \text{ is valid \iff } \forall (r,t) > \phi \]

---

**Expand**

**Embed (more) logic into your current programs**

Possible paths:

- **Translate existing code into a logic language**: e.g. Ada to Why3
  - The translation can be done mostly automated.
  - Some aspects can be proven automatically.

- **Expand/restrict** an existing language with stronger (real-time and concurrent) primitives
  (incl. contracts & invariants): e.g. Sparkel, Parasail
  - Many contracts can be proven automatically at compile time.

Some traditional language features should be or need to be avoided/removed:
- e.g. Aliasing (pointers), non-scoped or non-stack-based memory, exception handling.

---

**Summary**

**Reliability**

- **Terminology**
  - Faults, Errors, Failures - Reliability

- **Faults**
  - Faults, Fault avoidance, removal, prevention or Fault Tolerance.

- **Redundancy**
  - Static (TMR, NMR) and dynamic redundancy
  - Version programming, and dynamic-redundancy in software design.

- **Reduce & Formalise**
  - Ravenscar profile.
  - Real-time logic.