Reliability

Uwe R. Zimmer - The Australian National University

References for this chapter


(Schobbens99)

P.Y. Schobbens, JF Raskin, TA Henzinger, L. Ferrier

Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming

Draft 0.6, August 2013

(Lyu92)

Michael R. Lyu, Algirdas Avizienis

Sparkel Reference Manual

Deductive Program Verification


(Taft2013)

S. T. Taft

Predictable/dependable systems ...

... in the real-time domain!

Achieving reliability

© 2015 Uwe R. Zimmer, The Australian National University page 903 of 961 (chapter 1: “Reliability” up to page 949)
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).

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).
- Apply strict coding standards and target for code certification.
- Employ languages and runtime systems with reasonable support for the requirements.

Fault avoidance at development level:
- Team programming methods like extreme programming or rigorous testing!
- Investigate:
  - Static applications specifications.
  - Use an adequate (hardware) system design — Shock, humidity, interference, ...
  - Ensure proper assembly and encapsulation — Weak connections, bad PCBs, ...
  - Physical sensors and converters constraints.
  - Constraints of the employed controller network.
  - Constraints of the underlying run-time system.

Fault removal

Regardles of the rigor of fault prevention methods:
- Full fault tolerance of failures and the localization of faults.
- Graceful degradation (fail soft) 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.
- Fail safe the system halts and maintains its integrity.
- No re-evaluation method guarantees the total absence of faults.

Fault avoidance

Fault tolerance

Hardware redundancy

Fault tolerance

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).
- Apply strict coding standards and target for code certification.
- Employ languages and runtime systems with reasonable support for the requirements.

Fault avoidance at development level:
- Team programming methods like extreme programming or rigorous testing!
- Investigate:
  - Static applications specifications.
  - Use an adequate (hardware) system design — Shock, humidity, interference, ...
  - Ensure proper assembly and encapsulation — Weak connections, bad PCBs, ...
  - Physical sensors and converters constraints.
  - Constraints of the employed controller network.
  - Constraints of the underlying run-time system.

Fault removal

Find and remove errors from the previous stage.
- Team programming methods like extreme programming or rigorous testing!
- ... and more specifically for real-time and embedded systems.
- Tests can often not be performed under realistic conditions — especially exceptional conditions.
- Simulation environments frequently have a severe impact on real-time behaviour.
- The test space for real-time system is significantly larger than for non-real-time systems.

Fault tolerance

Hardware redundancy

Fault tolerance

Adding extra hardware resources:
- for the detection of failures and the localization of faults.
- for handling of exceptional situations and error-recovery.
- as a functional multiplication of complete sub-systems in order to be able to select the operational one in case of a failure in one part of the sub-system.

Fault-detection and recovery hardware includes:
- Watchdog timers, limit switches, additional physical sensors, transient-recording systems (emergency system dump), over-load-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 adds to the overall system complexity!

Fault avoidance

Fault tolerance

Hardware redundancy

Fault tolerance

Adding extra hardware resources:
- for the detection of failures and the localization of faults.
- for handling of exceptional situations and error-recovery.
- as a functional multiplication of complete sub-systems in order to be able to select the operational one in case of a failure in one part of the sub-system.

Fault-detection and recovery hardware includes:
- Watchdog timers, limit switches, additional physical sensors, transient-recording systems (emergency system dump), over-load-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 adds to the overall system complexity!

Fault avoidance

Fault tolerance

Hardware redundancy

Fault tolerance

Adding extra hardware resources:
- for the detection of failures and the localization of faults.
- for handling of exceptional situations and error-recovery.
- as a functional multiplication of complete sub-systems in order to be able to select the operational one in case of a failure in one part of the sub-system.

Fault-detection and recovery hardware includes:
- Watchdog timers, limit switches, additional physical sensors, transient-recording systems (emergency system dump), over-load-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 adds to the overall system complexity!

Fault avoidance

Fault tolerance

Hardware redundancy

Fault tolerance

Adding extra hardware resources:
- for the detection of failures and the localization of faults.
- for handling of exceptional situations and error-recovery.
- as a functional multiplication of complete sub-systems in order to be able to select the operational one in case of a failure in one part of the sub-system.

Fault-detection and recovery hardware includes:
- Watchdog timers, limit switches, additional physical sensors, transient-recording systems (emergency system dump), over-load-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 adds to the overall system complexity!

Fault avoidance

Fault tolerance

Hardware redundancy

Fault tolerance

Adding extra hardware resources:
- for the detection of failures and the localization of faults.
- for handling of exceptional situations and error-recovery.
- as a functional multiplication of complete sub-systems in order to be able to select the operational one in case of a failure in one part of the sub-system.

Fault-detection and recovery hardware includes:
- Watchdog timers, limit switches, additional physical sensors, transient-recording systems (emergency system dump), over-load-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 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 controller) were original system description documents (OSD) by Honeywell enhanced by additional cross-checking points and included some enhanced diversity elements in its language documents.
- 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.

### Table: Failure Rates and Error Detection

<table>
<thead>
<tr>
<th>Language</th>
<th>LOC</th>
<th>Test runs</th>
<th>Errors</th>
<th>Failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>2,296</td>
<td>512,000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1,330</td>
<td>—</td>
<td>—</td>
<td>568,318 × 10^-4</td>
</tr>
<tr>
<td>Modula-2</td>
<td>1,362</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>2,335</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
<tr>
<td>T (close to Lisp)</td>
<td>1,566</td>
<td>—</td>
<td>—</td>
<td>690,126 × 10^-4</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>1,913</td>
<td>—</td>
<td>—</td>
<td>627 × 10^-3</td>
</tr>
</tbody>
</table>

### Fault Tolerance

#### N-Version programming – Voting issues

**Integer arithmetic:**
- Integer (or any discrete-type) based results will be identical.
- N-Version programming will not help in this case.

**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.

**Multiple solutions:**
- The solution space itself allows for multiple correct, but structurally different solutions:
  - If there is a choice.
  - We specify the system.

### Dynamic Redundancy

#### Four constituent phases

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**
   - Frequency 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 need to be eliminated.

#### Damage confinement and assessment

**Error states from the environment:**
- Hardware, CPU, controller, communication systems, ...
- Environmental error state:
- Error states stemming from within the application processes:
  - Duplication – employ N-version programming to detect error states.
  - Timing, checking timers, and overrun detectors.
  - Reversal: apply the reverse function and compare.
  - Coding: detect corrupted data via redundant information (CRC-checks, ...).
  - Based on an approach:
    - Checking contracts (e.g. in Ada, Spark).
  - Structural: check structural integrity (e.g. lists, file systems).
  - Continuity: assuming a difference between consecutive controller values.

### Dynamic Redundancy

#### Error detection

**Confinement:**
- How to avoid the transfer of fault-effects between system parts?
  - Modular decomposition
  - Atomic actions
  - Firewall

**Assessment:**
- Identifying fault and its potential location:
  - Location of the detected error state.
  - All possible paths through the systems which are all leading to this error state.
  - A high-fan-in system structure (error-confinement) limits the length of the possible paths.
Reliability

Fault tolerance

Dynamic redundancy

Fault treatment

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?

Dependability features:
- Availability
- Reliability
- Safety
- Confi dentiality
- Integrity
- Maintainability

Further refinements in the design tool chain:

Restrict & Formalize

Restrict, Formalize, … ?

Restrict
- Limit the tools and environments to safer operations.

Formalize:
- Temporal logic, Real-Time Logic (RTL) as an extension of predicate logic.
- Classical real-time design and certification methods: MASCOT, JSI, MOOP, HMOD, HILDEWOO, COQNET, DO178B, …

Expand:

Reliability

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 sub-programs.
- No dynamic allocation or unchecked de-allocation of protected and task objects
- Tasks are assumed to be non-terminating
- no use of the select statement
- Library level Protected objects with no entries
- Barrier consisting of a single boolean variable

Ada Zero Footprint profile

No tasking
- no scheduling, no task switches, …
- No dynamic allocation
- no need for dynamic heap management.
- No dynamic deallocating
- no bindings happen at compile time.
- No exception propagation
- local exception handlers are still permitted.
- Packed arrays only pack component of component sizes of powers of two
- reduces code complexity.

Ada Ravenscar profile

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 response
- 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 structure against asynchronous task actions.
- No use of the select statement
- non-degradable behaviour is difficult to analyse, moreover the existence of protected objects has diminished the importance of the select statement to the tasking model.

Restrict

Ada Ravenscar profile

Restrict

No use of task entries
- no necessity to program systems that can be analysed; it follows that there is no need for the assert statement.
- “Only until” statement but no “delay” statement
- the absolute form of delay is not necessary to use for constructing periodic tasks
- to gain access to the real time clock.
- Atomic and Valuable packages
- needed to enforce the correct use of shared data.
- Grant attribute (but not within entry barriers)
- can be useful for some algorithms and has low overhead.

Reliability

Safety Terminology

Backward error recovery:
- Set checkpoints and safe the system state with each passing of a checkpoint.
- If an error state is detected, set back to the last consistent checkpoint.
- Applicable even if the fault location cannot be identified.
- Not applicable at all, if the system contains non-reversible components (times, …)

Forward error recovery:
- Method of choice for most time critical parts of real-time and embedded systems.
- May involve complex mode and priority changes (deadlines might be still relevant).

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

On-line fault treatment might be tricky and is usually limited to (hot) features:
- Availability — ready to use
- Reliability — absence of failures
- Safety — absence of lethal failures
- Confi dentiality — absence of unauthorized disclosures
- Integrity — no data corruptions
- Maintainability — accessibility to changes and improvements

Ada Zero Footprint profile

No dynamic allocation or unchecked de-allocation of protected and task objects
- No dynamic allocation or unchecked de-allocation of protected and task objects
- No use of the select statement
- Library level Protected objects with no entries
- Barrier consisting of a single boolean variable
- no side effects are possible and exit protocol becomes simple.
Reliability

Adapting the Ada Ravenscar profile
- Ada_task Identification
  - can be used for coarse-grained and has low overhead, available in restricted form (no Abort, Task, or Task attribute functions Callable or Terminated).
- Task discriminants
  - can be used 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.
- 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
  - are required for certain situations.

Ada Certification profiles
Profiles tailored to specific certification processes, e.g.
- DO-178B, DO-178C
  - can be useful for some algorithms and has low overhead, available in reduced form (no Abort, Task, or Task attribute functions Callable or Terminated).

Temporal logic

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

Linear Temporal Logic of Real Numbers (LTR)
\[ \phi = p(\phi_1 \lor \phi_2) \land \phi \lor \phi_3 \land \phi_4 \]

where:
\[ m(x) = \left\{ \begin{array}{ll}
1 & \text{if } x \in E(x) \\
0 & \text{otherwise}
\end{array} \right. \]

Interpretation: does \( m(x) \) denote all possible, all defined, or all observed instances of \( E(x) \)

Event-Clock Temporal Logic
\[ \phi = p(\phi_1 \lor \phi_2) \land \phi \lor \phi_3 \land \phi_4 \land \phi_5 \]

where:
\[ m(x) = \left\{ \begin{array}{ll}
1 & \text{if } x \in E(x) \\
0 & \text{otherwise}
\end{array} \right. \]

Interpretation: does \( m(x) \) denote all possible, all defined, or all observed instances of \( E(x) \)

Real-Time Logic

Assertions on sequences and orders of states:
\( \Delta A \) is true for all future states.
\( \Delta A \) is eventually true.
\( \Delta A \) is true for the following state.

Examples:
\( \text{Collision_Warning } = \text{Collision_Assurance} \)
\( \text{Collision_Warning } = \text{Collision_Assurance} \)
assuming that there is a sequence of distinguishable states (or time).

Formalize

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.

Examples:
\[ \forall t \in E(x) \ldots \]

Interpretations: does \( m(x) \) denote all possible, all defined, or all observed instances of \( E(x) \)
Formalize

Metric-Interval Temporal Logic

\[ \phi_i = p \land (\phi_i^0 \lor \phi_i^1) \lor (\phi_i^2 \land \phi_i^3) \]

where:

\[ (r,t) \models p \quad \text{iff} \quad p \in I(r,t) \]
\[ (r,t) \models (\phi_1 \lor \phi_2) \quad \text{iff} \quad (r,t) \models \phi_1 \text{ or } (r,t) \models \phi_2 \]
\[ (r,t) \models \neg \phi \quad \text{iff} \quad (r,t) \not\models \phi \]
\[ (r,t) \models (\phi_1 \land \phi_2) \quad \text{iff} \quad \exists ! \in I(r,t) \mid \exists t' \in I(r,t') \mid t' \models \phi_1 \text{ and } t' \models \phi_2 \]
\[ (r,t) \models \phi_i \quad \text{iff} \quad \exists t' \in I(r,t') \mid t' \models \phi_i \]

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

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 avoided or replaced:
- Aliasing (pointers), non-scoped or non-stack-based memory exception handling

Summary

Reliability

Terminology
- Faults, Errors, Failures – Reliability
- Fault avoidance, removal, prevention – Fault tolerance

Redundancy
- Static (NMR, TMR) and dynamic redundancy
- N-version programming, and dynamic redundancy in software design

Reduce & Formalise
- Ravenscar profile
- Real-time logic