Reliability, failure & tolerance

Based on a set of powerful and diverse tools ...

... reconsider 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!

References for this chapter

[Burns98] Alan Burns, Brian Dobbing, George Romanski
The Ravenscar Tasking Profile for High Integrity Real-Time Programs

[Filliatre2013] J.-C. Filliatre
Deductive Program Verification with Why3 – A Tutorial
Lecture at DigiCosme Spring School 2013

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

Axioms for Real-Time Logic

[Taft2013] S. T. Taft
Sparkel Reference Manual
Draft 0.6, August 2013
Reliability, failure & tolerance

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 logic domain

- Non-termination / -completion
  - Systems 'frozen' in a deadlock state, blocked for missing input, or in an infinite loop
  - Watchdog timers required to handle the failure
- Range violations and other inconsistent states
  - Run-time environment level exception handling required to handle the failure
- Value violations and other wrong results
  - User-level exception handling required to handle the failure

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

Observable failure modes
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 PCBs, …

Fault avoidance at software design level:
- Verify consistency of specifications (employ formal methods where applicable).
- Employ automated deduction (theorem provers) at compile-time.
- Apply strict coding standards and target for code certification.
- 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?
- No re-evaluation method guarantees the total absence of faults.

… 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 behaviours.
- 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 actual real-time system might still fail!

This is specifically critical for unmonitored systems:
- Systems which are (temporary) inaccessible.
- Un-manned vehicles which operate semi-autonomously by default.
- Systems in remote / hostile environments.

Fault tolerance

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 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.

Any hardware redundancy adds to the overall system complexity!

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.
- 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.

The outputs from the different parts will be slightly different …
Fault tolerance

**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 built 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 or cause more than the loss of one primary flight computer.

Sophisticated synchronization and communication systems.

(so far no fatal event due to avionics failure — information from November 2013)

---

**N-version programming**

Impacts to software diversity:

<table>
<thead>
<tr>
<th>Development</th>
<th>Languages</th>
<th>Tools</th>
<th>Algorithms</th>
<th>Methodologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Language</th>
<th>L.o.c.</th>
<th>Test runs</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>1531</td>
<td>—</td>
<td>568</td>
<td>$1.088 \times 10^{-4}$</td>
</tr>
<tr>
<td>Modula-2</td>
<td>1562</td>
<td>—</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pascal</td>
<td>2331</td>
<td>—</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>2228</td>
<td>—</td>
<td>680</td>
<td>$1.326 \times 10^{-4}$</td>
</tr>
<tr>
<td>T (close to Lisp)</td>
<td>1568</td>
<td>—</td>
<td>680</td>
<td>$1.326 \times 10^{-4}$</td>
</tr>
<tr>
<td>Average</td>
<td>1913</td>
<td>—</td>
<td>321</td>
<td>$6.27 \times 10^{-5}$</td>
</tr>
</tbody>
</table>
"The six-language project"

<table>
<thead>
<tr>
<th>Failure category</th>
<th>Average probability 10^-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors</td>
<td>6.27 × 10^-5</td>
</tr>
<tr>
<td>Single error</td>
<td>13.05 × 10^-5</td>
</tr>
<tr>
<td>Two distinct errors</td>
<td>0.20 × 10^-5</td>
</tr>
<tr>
<td>Two coincident errors</td>
<td>2.65 × 10^-5</td>
</tr>
<tr>
<td>Three errors</td>
<td>0.34 × 10^-5</td>
</tr>
</tbody>
</table>

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

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 faults.

N-version programming – Voting issues

Integer arithmetic:
- Integer (or any discrete-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. ✓

Multiple solutions:
- The solution space itself allows for multiple correct, but structurally different solutions:
  - ✖ re-specify the system

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 a similar level of reliability.
Fault tolerance

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.

Error states from the environment:
- Hardware… CPU, controllers, communication systems, ...
- Run-time environment.

Error states stemming from checks within 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.
- Coding – detect corrupted data via redundant information (CRC-checks,...).
- Reasonableness – check contracts (e.g. in Eiffel, Ada, Spark).
- Structural – check structural integrity (e.g. lists, file-systems).
- Continuity – assuming a limited difference between consecutive controller values.
- ...

Backward error recovery:
- Set checkpoints and save the system state with each passing of a checkpoint. How can system-wide consistent checkpoints be ensured?
- If an 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 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).
Restrict & Formalize

Further refinements in the design tool chain:

Restrict, Formalize, ... ?

Restrict:
- Limit the tools and environments to ‘safer’ operations.
  
  e.g. Esterel, High-Integrity Pearl, Ada Ravenscar profile, Ada DO178B profile/runtime...

Formalize:
- Temporal logic, Real-Time Logic (RTL) as an extension of predicate logic.
- Classical real-time design and certification methods: MASCOT, JSD, MOON, HOOD, HRT-HOOD, CODARTS, DO178B, ...

Expand:
- Expand a language by means of provable predicates: SPARK, Sparkel, Parasail, Why3, ...

Fault tolerance

Dynamic redundancy

fault treatment

Adjust, avoid, correct, or exchange:

- 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 usually an expensive and complex operation.

⇒ The number of substitutable sub-systems in a dynamic redundant system is limited.
  (Many systems will assume transient faults, log the event and continue operations ... )

Fault tolerance

Dynamic redundancy

fault treatment

Adjust, avoid, correct, or exchange:

- 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 usually an expensive and complex operation.

⇒ The number of substitutable sub-systems in a dynamic redundant system is 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?

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

Terminology

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 — ready to use
- Reliability — absence of failures
- Safety — absence of fatal failures
- Confidentiality — absence of unauthorized disclosures
- Integrity — no data corruptions
- Maintainability — accessibility to changes and improvements

Ada Zero Footprint profile

- No tasking.
  ⇒ no scheduling, no task switches, ...

- No dynamic allocation.
  ⇒ removes the need for dynamic heap management.

- No dynamic dispatching.
  ⇒ all bindings happen at compile time.

- No controlled types.
  ⇒ clean ups have to be done in static scopes.

- 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

- 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 overhead of an exit protocol.
- Barrier consisting of a single boolean variable.
  - No side effects are possible and exit protocol becomes simple.

- 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.
- "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 only (no Calendar package).
  - To gain access to the real-time clock.
- 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.
Temporal Logic

- Extending predicate logic.
- Adding a concepts of ordering for events and states.
- Suitable for event driven system, reactive systems
  - Esterel, Times CSP, ...

Formalize

Temporal Logic

Restrict

Ada Ravenscar profile

- 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.
- 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.

Restrict

Ada Certification profiles

Profiles tailored to specific certification processes, e.g.

DO-178B, DO-178C

managed by the Radio Technical Commission for Aeronautics (RTCA).

which are used by e.g. the

Federal Aviation Administration (FAA)
European Aviation Safety Agency (EASA)

Addresses e.g.:

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

Formalize

Temporal Logic

Assertions on sequences and orders of states
- employ predicate logic & a set of new operators:
  - □A: A is true for all future states.
  - ◇A: A is eventually true.
  - ◦A: A is true for the following state.

Examples:

□(Collision_Warning ⇒ ◇Collision_Avoidance)
□(Collision_Warning ⇒ ◦Collision_Avoidance)

assuming that there is a sequence of distinguishable states (or ‘time’).
Formalize

Temporal Logic

Another common temporal logic operator ("Until"):

Example:

\[
\square ((\text{Tasks}_\text{Waiting} \land \text{Entry}_\text{Open}) \land (\neg \text{Tasks}_\text{Waiting} \land \text{Entry}_\text{Closed}))
\]

Note:
- Temporal logic expresses the order of events only and has no means to express explicit temporal scopes, deadlines, etc.
- Explicit temporal specifications need to be translated into event relations first.

Formalize

Real-Time Logic

Occurrence times of predicates:

\[
\downarrow A \text{ denotes the time when } A \text{ changes from false to true.}
\]

\[
\downarrow A \text{ denotes the time when } A \text{ changes from true to false.}
\]

Examples:

\[
\forall i \exists j \left( (\exists (E,i) \leq (\downarrow A,j)) \land (\forall (i, A,j) \leq (E,i) + d) \land (\forall (i, A,j + 1) \leq (E,i) + p) \right)
\]

Interpretations: does \( \forall i \exists (E,i) \) denote all possible, all defined, or all observed instances of \( E \)?

Formalize

Linear Temporal Logic of Real Numbers (LTR)

\[
\phi := p \mid \phi_1 \lor \phi_2 \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \phi_2
\]

where:

\[
(\tau, t) \vdash p \quad \text{iff} \quad p \in \tau(t)
\]

\[
(\tau, t) \vdash \phi_1 \lor \phi_2 \quad \text{iff} \quad (\tau, t) \vdash \phi_1 \text{ or } (\tau, t) \vdash \phi_2
\]

\[
(\tau, t) \vdash \neg \phi \quad \text{iff} \quad (\tau, t) \nvdash \phi
\]

\[
(\tau, t) \vdash \phi_1 \land \phi_2 \quad \text{iff} \quad \exists \tau' > \tau \tau ' \vdash \phi_2 \text{ and } \forall \tau ' \in (\tau, \tau) \tau ' \vdash \phi_1 \land \phi_2
\]

\[
(\tau, t) \vdash \phi_1 \phi_2 \quad \text{iff} \quad \exists \tau' > \tau \tau ' \vdash \phi_2 \text{ and } \forall \tau ' \in (\tau, \tau) \tau ' \vdash \phi_1 \lor \phi_2
\]

\( \phi \) is satisfiable iff \( \exists (\tau, t) \vdash \phi \)

\( \phi \) is valid iff \( \forall (\tau, t) \vdash \phi \)
### Event-Clock Temporal Logic

\[ \phi := p \mid \phi_1 \lor \phi_2 \mid \neg \phi \mid \phi_1 \cup \phi_2 \mid \phi_1 \land \phi_2 \mid < t \mid > t \phi \]

where:

\[
\begin{align*}
(\tau, t) > p & \quad \text{iff} \quad p \in \tau(t) \\
(\tau, t) > \phi_1 \lor \phi_2 & \quad \text{iff} \quad (\tau, t) > \phi_1 \text{ or } (\tau, t) > \phi_2 \\
(\tau, t) > \neg \phi & \quad \text{iff} \quad (\tau, t) \not> \phi \\
(\tau, t) > \phi_1 \cup \phi_2 & \quad \text{iff} \quad \exists t' > t \land t' \not< t \land t' \not< t' \land \forall t'' \in (t, t')] [t''] \not< t' \land \phi_1 \lor \phi_2 \\
(\tau, t) > \phi_1 \land \phi_2 & \quad \text{iff} \quad \exists t' > t \land t' \not< t \land t' \not< t' \land \forall t'' \in (t, t')] [t''] \not< t' \land \phi_1 \land \phi_2 \\
(\tau, t) > < t \phi & \quad \text{iff} \quad \exists t' > t \land t' \not< t \land t' \not< t' \land \forall t'' \in (t, t')] [t''] \not< t' \land \phi \\
(\tau, t) > > t \phi & \quad \text{iff} \quad \exists t' > t \land t' \not< t \land t' \not< t' \land \forall t'' \in (t, t')] [t''] \not< t' \land \phi \\
\phi \text{ is satisfiable iff } \exists (\tau, t) \models \phi \\
\phi \text{ is valid iff } \forall (\tau, t) \models \phi
\end{align*}
\]

### Metric-Interval Temporal Logic

\[ \phi := p \mid \phi_1 \lor \phi_2 \mid \neg \phi \mid \phi_1 \cup \phi_2 \mid \phi_1 \land \phi_2 \mid < t \mid > t \phi \]

where:

\[
\begin{align*}
(\tau, t) > p & \quad \text{iff} \quad p \in \tau(t) \\
(\tau, t) > \phi_1 \lor \phi_2 & \quad \text{iff} \quad (\tau, t) > \phi_1 \text{ or } (\tau, t) > \phi_2 \\
(\tau, t) > \neg \phi & \quad \text{iff} \quad (\tau, t) \not> \phi \\
(\tau, t) > \phi_1 \cup \phi_2 & \quad \text{iff} \quad \exists t' \in (t, t'] [t''] \not< t' \land \phi_1 \lor \phi_2 \\
(\tau, t) > \phi_1 \land \phi_2 & \quad \text{iff} \quad \exists t' \in (t, t'] [t''] \not< t' \land \phi_1 \land \phi_2 \\
\phi \text{ is satisfiable iff } \exists (\tau, t) \models \phi \\
\phi \text{ is valid iff } \forall (\tau, t) \models \phi
\end{align*}
\]

### 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 feature should be or need to be avoided/replaced:

- e.g. Aliasing (pointers), non-scoped or non-stack-based memory, exception handling, ...