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!
Reliability

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
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 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.

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

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
Reliability

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.
  - or in case of a detected error-condition: dynamic parts which are swapped in.

☞ 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)
Impacts to software diversity:

<table>
<thead>
<tr>
<th>Development teams</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.

### Language Sources (l.o.c.) Test runs Errors Failure-rate

<table>
<thead>
<tr>
<th>Language</th>
<th>Sources (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>1.108×10⁻⁴</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>1.326×10⁻⁴</td>
</tr>
<tr>
<td>T (close to Lisp)</td>
<td>1568</td>
<td></td>
<td>680</td>
<td>1.326×10⁻⁴</td>
</tr>
<tr>
<td>Average</td>
<td>1913</td>
<td></td>
<td>321</td>
<td>0.627×10⁻⁴</td>
</tr>
</tbody>
</table>
Real-Time & Embedded Systems

Reliability

“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
    ➩ Solution space itself allows for multiple correct, but different solutions
    ➩ Independence ☒ 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 coincident 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 \( x \Rightarrow f^{-1}(f(x)) \)
  - 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.
  - ...

**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 — Damage diagnosis**

**Dynamic redundancy — Error recovery**

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 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 ...)
Reliability

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
  - Maintainability — accessibility to changes and improvements

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.

… 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)

- **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.

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.

Temporal logic

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

Temporal logic

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

  \[ \Box A: \text{A is true for all future states} \]
  \[ \Diamond A: \text{A is eventually true} \]
  \[ \lozenge A: \text{A is true for the following state} \]

  e.g. \[ \Box (\text{Collision Warning} \Rightarrow \Diamond \text{Collision Avoidance}) \]
  or: \[ \Box (\text{Collision Warning} \Rightarrow \lozenge \text{Collision Avoidance}) \]

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

Temporal logic

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

e.g.
$((\text{Tasks\_Waiting} \ \mu \ \text{Entry\_Closed}) \land \neg(\text{Tasks\_Waiting} \ \mu \ \text{Entry\_Open}))$

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

Real-Time Logic

• Assertions on real-time events:
  
  $\Rightarrow$ employ predicate logic & a occurrence function:
  
  $(E, i)$: denotes the time of the $i$-th occurrence of event (-class) $E$
  
  $\Rightarrow$ the event (-class) $E$ is strictly ordered by instance ($i$) and time ($\tau$).
  
  $\Rightarrow$ all events of kind (class) $E$ can be distinguished.
  
  instance order $\Rightarrow$ order in time.

Occurrence times of predicates:
$
\Up:\$ denotes the time when $A$ changes from false to true.
$
\Down:\$ denotes the time when $A$ changes from true to false.

Examples:
$
\forall i, j ((\up(E, i) < \down(A, j) \land \down(A, j) \leq (\down(A, j-1) \leq (\up(E, i)) \land \forall i ((\up(E, i+1) \geq (\down(E, i)) + p))
\forall i, j ((\up(A, i) < \down(B, j) \lor (\down(B, j) < \up(A, i))))$

Linear Temporal Logic of Real Numbers (LTR)

$$
\phi ::= p | \phi_1 \lor \phi_2 | \neg \phi_1 | U \phi_2 | S \phi_2
$$

where
$(\tau, t) \vdash p$ iff $p \in \tau(t)$
$(\tau, t) \vdash \phi_1 \lor \phi_2$ iff $(\tau, t) \vdash \phi_1$ or $(\tau, t) \vdash \phi_2$
$(\tau, t) \vdash \neg \phi$ iff $(\tau, t) \not\vdash \phi$
$(\tau, t) \vdash \phi_1 U \phi_2$ iff $\exists \tau' > t \land \tau' \not\vdash \phi_2$ and $\forall \tau'' \in (t, \tau')$, $\tau'' \not\vdash \phi_1 \lor \phi_2$
$(\tau, t) \vdash \phi_1 S \phi_2$ iff $\exists \tau' < t \land \tau' \not\vdash \phi_2$ and $\forall \tau'' \in (t, \tau')$, $\tau'' \not\vdash \phi_1 \lor \phi_2$

$\phi$ is satisfiable iff $\exists (\tau, t) \vdash \phi$ — $\phi$ is valid iff $\forall (\tau, t) \vdash \phi$
Reliability

Event-Clock Temporal Logic

where

\( \phi := p | \phi_1 \lor \phi_2 | \neg \phi_1 | U_{\phi_2} | S_{\phi_2} | < I_{\phi} | > I_{\phi} \)

\( (\tau, t) \vdash p \iff p \in \tau(t) \)

\( (\tau, t) \vdash \phi_1 \lor \phi_2 \iff (\tau, t) \vdash \phi_1 \) or \( (\tau, t) \vdash \phi_2 \)

\( (\tau, t) \vdash \neg \phi \iff (\tau, t) \nvDash \phi \)

\( (\tau, t) \vdash \phi_1 U_{\phi_2} \iff \exists t' > t \land t' \vdash \phi_2 \) and \( \forall t'' \in (t, t'), t'' \nvDash \phi_1 \lor \phi_2 \)

\( (\tau, t) \vdash \phi_1 S_{\phi_2} \iff \exists t' < t \land t' \vdash \phi_2 \) and \( \forall t'' \in (t', t), t'' \nvDash \phi_1 \lor \phi_2 \)

\( (\tau, t) \vdash < I_{\phi} \iff \exists t' < t \land t' \vdash \phi \) and \( \forall t'' \in (t - l, t), t'' \nvDash \phi \)

\( (\tau, t) \vdash > I_{\phi} \iff \exists t' > t \land t' \vdash \phi \) and \( \forall t'' \in (t, t + l), t'' \nvDash \phi \)

Metric-Interval Temporal Logic

where

\( \phi := p | \phi_1 \land \phi_2 | \neg \phi_1 \hat{U}_{\phi_2} | \hat{S}_{\phi_2} \)

\( (\tau, t) \vdash p \iff p \in \tau(t) \)

\( (\tau, t) \vdash \phi_1 \land \phi_2 \iff (\tau, t) \vdash \phi_1 \) and \( (\tau, t) \vdash \phi_2 \)

\( (\tau, t) \vdash \neg \phi \iff (\tau, t) \nvDash \phi \)

\( (\tau, t) \vdash \phi_1 \hat{U}_{\phi_2} \iff \exists t' \in (t, t + l) \land t' \vdash \phi_2 \) and \( \forall t'' \in (t', t), t'' \nvDash \phi_1 \)

\( (\tau, t) \vdash \phi_1 \hat{S}_{\phi_2} \iff \exists t' \in (t - l, t) \land t' \vdash \phi_2 \) and \( \forall t'' \in (t', t), t'' \nvDash \phi_1 \)