Real-Time & Embedded Systems 2019

9

Reliability

Uwe R. Zimmer - The Australian National University

Reliability

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)

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

[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.), IEEEComputer Society Press Technology Series, pp. 45-54, October (1992)

[Schobbens99]
P.Y Schobbens, J.F Raskin, T.A Henzinger, L.Ferier
Axioms for Real-Time Logic
Lecture Notes in Computer Science 1466, Springer-Verlag, 1999, pp. 219-236 (2002)

[Taft2013]
S. T. Taft
Sparkel Reference Manual
Draft 0.6, August 2013

Reliability

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!

Reliability

Reliability, failure & tolerance

'Terminology of failure' or 'Failing terminology'?

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

::= low failure rate.

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

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

Fault ::= the reason for an error.

Reliability

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

Reliability

Reliability, failure & tolerance

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

---

Reliability

Reliability, failure & tolerance

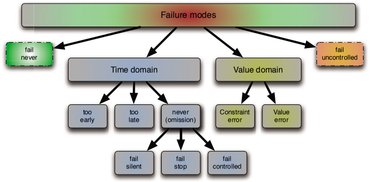
Faults in the time domain

- 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

Reliability

Reliability, failure & tolerance

Observable failure modes



Reliability

Reliability

Achieving reliability

Reliability

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).

☞ To understand all critical real-time requirements and issues.

Reliability

Reliability

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 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.

Reliability

Reliability

Fault removal

Find and remove errors from the previous stage.

☞ Team programming methods like extreme programming or rigorous testing?

yet …

☞ 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.

Reliability

Reliability, failure & tolerance

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**

Reliability

Reliability, failure & tolerance

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 'foreseeable' 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.

Reliability

Fault tolerance

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 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!

Reliability

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 …

Reliability

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 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: $< {}^{10^{-10}}\!/_{\mathrm{h}}$ (e.g. UK Seizewell B nuclear reactor (emerg.): $< {}^{10^{-3}}\!/_{\mathrm{h}}$)

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.

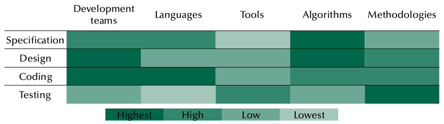
(so far no fatal event due to avionics failure — information from November 2013)

Reliability

Fault tolerance

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

Source: [Lyu92]

Reliability

Fault tolerance

"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.

Reliability

Fault tolerance

"The six-language project"

Language	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	6.27×10^{-5}

Reliability

Fault tolerance

"The six-language project"

Failure category	Average … Single version … failure probabilities measured in … 5,127,400 cases	Average … 3-version … failure probabilities measured in … 102,531,685 cases	Average … 5-version … failure probabilities measured in … 30,757,655 cases
No errors	0.99993733	0.9998409	0.9997807
Single error	6.27×10^{-5}	13.05×10^{-5}	19.15×10^{-5}
Two distinct errors		0.20×10^{-5}	0.23×10^{-5}
Two coincident errors		2.65×10^{-5}	2.21×10^{-5}
Three errors			0.34×10^{-5}

Reliability

Fault tolerance

"The six-language project"

☞ 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 faults.

Reliability

Fault tolerance

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

Reliability

Fault tolerance

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.

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.

Reliability

Fault tolerance

Dynamic redundancy

Error detection

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.
- …

Reliability

Fault tolerance

Dynamic redundancy

Damage confinement and assessment

Confinement:

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

- Modular decomposition
- Atomic actions
- 'Firewalls'

Assessment:

☞ Identifying fault and its (potential) location:

- Location of the detected error state.
- All 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.

Reliability

Fault tolerance

Dynamic redundancy

Error recovery

Backward error recovery:

- Set checkpoints and safe the system state with each passing of a check-point. 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 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).

Reliability

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 …)

Reliability

Terminology

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

Reliability

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, …

Reliability

Restrict

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.

Reliability

Restrict

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.

Reliability

Restrict

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.

Reliability

Restrict

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 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.

Reliability

Restrict

Ada Ravenscar profile

- **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.

Reliability

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.

Reliability

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

Reliability

Formalize

Temporal Logic

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

☞ Esterel, Times CSP, …

Reliability

Formalize

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*.

Examples:
$\Box$(Collission_Warning $\Rightarrow$ $\Diamond$Collission_Avoidance)
$\Box$(Collission_Warning $\Rightarrow$ $\bigcirc$Collission_Avoidance)
assuming that there is a sequence of distinguishable states (or 'time') .

Reliability

Formalize

Temporal Logic

Another common temporal logic operator ("Until"):

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

Example:

$\Box$ ((Tasks_Waiting μ Entry_Open) $\wedge$ ($\neg$Tasks_Waiting μ Entry_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.

Reliability

Formalize

Real-Time Logic

Assertions on real-time events:

☞ Employ predicate logic & an occurrence function:

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

Assumptions:

☞ The event (-class) E is strictly ordered by instance (i) and time ($@$).
☞ All events of kind (class) E can be distinguished.
☞ Instance order ⇒ order in time.

Reliability

Formalize

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:

$$\forall i \exists j \begin{pmatrix} (@(E,i) \le @(\uparrow A,j)) \\ \wedge(@(\downarrow A,j) \le @(E,i)+d) \\ \wedge(@(\downarrow A,j-1) \le @(E,i)) \\ \wedge \forall i(@(E,i+1) \ge @(E,i)+p) \end{pmatrix} \qquad \forall i,j \begin{pmatrix} (@(\downarrow A,i) < @(\uparrow B,j)) \\ \vee(@(\downarrow B,j) < @(\uparrow A,i)) \end{pmatrix}$$

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

Reliability

Formalize

Linear Temporal Logic of Real Numbers (LTR)

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

where:

$$\begin{array}{lll} (\tau,t) \vDash p & \text{iff} & p \in \tau(t) \\ (\tau,t) \vDash \phi_1 \vee \phi_2 & \text{iff} & (\tau,t) \vDash \phi_1 \text{ or } (\tau,t) \vDash \phi_2 \\ (\tau,t) \vDash \neg\phi & \text{iff} & (\tau,t) \nvDash \phi \\ (\tau,t) \vDash \phi_1 \mathrm{U} \phi_2 & \text{iff} & \exists t' > t \mid t' \vDash \phi_2 \text{ and } \forall t'' \in (t,t') \mid t'' \vDash \phi_1 \vee \phi_2 \\ (\tau,t) \vDash \phi_1 \mathrm{S} \phi_2 & \text{iff} & \exists t' < t \mid t' \vDash \phi_2 \text{ and } \forall t'' \in (t',t) \mid t'' \vDash \phi_1 \vee \phi_2 \end{array}$$

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

Reliability

Formalize

Event-Clock Temporal Logic

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

where:

$$\begin{array}{lll} (\tau,t) \vDash p & \text{iff} & p \in \tau(t) \\ (\tau,t) \vDash \phi_1 \vee \phi_2 & \text{iff} & (\tau,t) \vDash \phi_1 \text{ or } (\tau,t) \vDash \phi_2 \\ (\tau,t) \vDash \neg\phi & \text{iff} & (\tau,t) \nvDash \phi \\ (\tau,t) \vDash \phi_1 \mathrm{U} \phi_2 & \text{iff} & \exists t' > t \mid t' \vDash \phi_2 \text{ and } \forall t'' \in (t,t') \mid t'' \vDash \phi_1 \vee \phi_2 \\ (\tau,t) \vDash \phi_1 \mathrm{S} \phi_2 & \text{iff} & \exists t' < t \mid t' \vDash \phi_2 \text{ and } \forall t'' \in (t',t) \mid t'' \vDash \phi_1 \vee \phi_2 \\ (\tau,t) \vDash \lhd_I \phi & \text{iff} & \exists t' < t \mid t' \in t - I \wedge t' \vDash \phi \text{ and } \forall t'' \in (t-I,t) \mid t'' \vDash \phi \\ (\tau,t) \vDash \rhd_I \phi & \text{iff} & \exists t' > t \mid t' \in t + I \wedge t' \vDash \phi \text{ and } \forall t'' \in (t,t+I) \mid t'' \vDash \phi \end{array}$$

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

Reliability

Formalize

Metric-Interval Temporal Logic

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

where:

$$\begin{array}{lll} (\tau,t) \vDash p & \text{iff} & p \in \tau(t) \\ (\tau,t) \vDash \phi_1 \vee \phi_2 & \text{iff} & (\tau,t) \vDash \phi_1 \text{ or } (\tau,t) \vDash \phi_2 \\ (\tau,t) \vDash \neg\phi & \text{iff} & (\tau,t) \nvDash \phi \\ (\tau,t) \vDash \phi_1 \hat{\mathrm{U}}_I \phi_2 & \text{iff} & \exists t' \in (t,t+I) \mid t' \vDash \phi_2 \text{ and } \forall t'' \in (t,t') \mid t'' \vDash \phi_1 \\ (\tau,t) \vDash \phi_1 \hat{\mathrm{S}}_I \phi_2 & \text{iff} & \exists t' \in (t-I,t) \mid t' \vDash \phi_2 \text{ and } \forall t'' \in (t',t) \mid t'' \vDash \phi_1 \end{array}$$

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

Reliability

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 feature should be or need to be avoided/replaced:
e.g. Aliasing (pointers), non-scoped or non-stack-based memory, exception handling, …

Reliability

Summary

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**
  - Ravenscar profile.
  - Real-time logic.