IV. Foundations

IV.1 Specifications & Models


**model**
A simplified representation of something (the referent). The representation may be physical or abstract, and may be restricted to certain properties of the referent. In computing, models are usually abstract and are typically represented in a diagramming notation, such as *dataflow diagrams*, *ERA diagrams* (for a data model), or *state-transition diagrams* (for a model of behavior).

**specification**
A formal description of a system, or a component or module of a system, intended as a basis for further development. The expression of the specification may be in text in a natural language (e.g. English), in a *specification language*, which may be a formal mathematical language, and by the use of specification stages of a methodology that includes a diagrammatic technique. Characteristics of a good specification are that it should be unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable after development.

**model-based specification**
A form of specification, usually software specification, that is developed by creating a mathematical model of that system. Typically the mathematical model is expressed in terms of objects and operations, and these are derived using such mathematical concepts as sets, relations and functions.

**specification language**
A language, often a machine-processible combination of natural and formal language, used to express the requirements, design, behavior, or other characteristics of a system or component. For example, a design language or requirements specification language.

Contrast with: programming language; query language.

[IEEE-610.12-1990]

**formal language**
A language whose rules are explicitly established prior to its use. Examples include programming languages and mathematical languages. Contrast with: natural language.

[IEEE-610.12-1990]

**formal specification**
(1) A specification written and approved in accordance with established standards.

(2) A specification written in a formal notation, often for use in proof of correctness.

[IEEE-610.12-1990]
My Definitions

- What is a model?
  A model is an abstract description of an artefact that reflects all for a given problem/perspective relevant details.

- What is a specification?
  A specification is a description of the required properties that reflects all for a given problem/perspective all relevant details an artefact has to ensure.

Models vs. Specifications

- Model M:
  abstract description of an artefact

- Specification φ:
  a description of required properties an artefact has to ensure

Relation (usually only considered for formal ...):

- A model can fulfill a specification: M ⊦ φ
- A model M is also a specification (M ⊦ φ : M ⊦ φ ⇒ M ⊦ φ)
- All models fulfill the empty specification: M ⊦ true
- A given specification φ can be inconsistent and no model fulfilling it exists: ¬∃ M: M ⊦ φ

Formal Models

- What is a conceptual model?
  A conceptual model is an abstract description of an artefact that reflects all for a given problem/perspective relevant details.

- What is a formal model?
  A formal model is an abstract description with defined syntax and semantics for an artefact that reflects all for a given problem/perspective relevant details.

Formal Specification

- What is an informal specification?
  An informal specification is a set of required properties that reflects all for a given problem/perspective all relevant details an artefact has to ensure.

- What is a formal specification?
  A formal specification is a set of required properties with defined syntax and semantics that reflects all for a given problem/perspective all relevant details an artefact has to ensure.

Formal Models & Specifications (1/2)

Different kind of per Model/specification analysis:

- The model is syntactically correct
  no typing errors, ...

- The model is semantically correct
  General requirements (e.g., no deadlocks)
  The specification is syntactically correct
  no syntax errors, ...
  The specification is semantically correct
  not inconsistent, ...

Formal Models & Specifications (2/2)

Different kind of analysis possible (formal methods):

- Model & specification
  - model fulfills specification (M ⊦ φ)

- Model & model
  - Refinement (M ⊆ M): M is a refinement of M
    (Abstraction (M ⊆ M): M is a abstraction of M)
    M ⊆ M implies ∀ φ : (M ⊦ φ → M ⊦ φ) where the considered set of all specifications φ determines what notion of refinement is required

- Specification & specification
  - Refinement (φ ⇔ φ): φ is stronger than φ
    (Abstraction (φ ⇔ φ): φ is weaker than φ)
Informal Specifications

Why not formal?
- The intention of an informal specification is often to foster communication between the customer and the requirement engineer.
- The mental distance between the mental model of the modeller and the formalism is often rather large:
  - Efforts to do a correct formalization are often costly.
  - A formalization can result in subtle specification errors (which are difficult to detect).

Examples for informal specifications in SWE:
- requirements documents
- Specification documents

Conceptual Models

Why not formal?
- The intention of the model is to understand rather than determine all “details”.
- Misunderstanding about underspecification (see refinement).
- Modelling at the right level of abstraction is a complex task.
- Efforts to do a correct formalization are often too high.

Examples for conceptual models in SWE:
- OO analysis models (no semantics!)
- Architectural drawings
- UML today often a formal interpretation is possible!

IV.1.1 Fault Models

A fault model determines which possible effects of faults on the behaviour of a system model are considered.

- Usually explicit considered only for random hardware faults.
- Often no perfect representation of all possible physical effects.
- But can assist in the simulation of possible fault, the design of fault tolerant components, …

Single Stuck-at Fault Model

Assumptions:
- Basic hardware function seen as “black-box”.
- Fault modelled as:
  - input or output error;
  - stuck at either 1 or 0.
- Models only permanent faults.

Hardware Fault Models

Fault Models:
- Single-stuck-at fault model
- Bridging fault model
- Stuck-open fault model (CMOS)

General Concept:
- derive faults by altering system model

Similar concept for software:
- Mutation testing includes a similar approach, because the faults are assumed to be simple changes of the program code (no assumption that these are all possible faults).
Problem of Software Fault Models

Hardware:
- Physical defects affect only a part of the system
  - One mechanical component
  - limited area on a chip
- alter only one logic element

Software:
- Changed variable can possibly affect the correctness of any routine of a component accessing it (only limited by encapsulation)
- Corrupted op-code can result in arbitrary behaviour of the whole component

⇒ Consider software modules

Faults of Software Modules

Possible component faults:
- Crash (implicit fail silent assumption)
- Corrupted behaviour with maximal duration (watchdog)
- Corrupted behaviour without duration limit

Possible communication faults:
- Reordering of messages
- Lost massages
- Corrupted messages
- Repeated messages
- Arbitrary faked messages (Byzantine faults)

agreement for 3N+1 works only when at most N cheat!

Fault Models: Discussion

Use a coarse grain fault model:
+ Efficient algorithms for test pattern generation
+ Cheap design techniques for fault tolerance
- Relevant faults are missed

More fine grain fault model:
+ More relevant faults are covered
- Only inefficient algorithms for test pattern generation
- Only expensive design techniques for fault tolerance
- Sometimes too much and complex faults have to be considered

Fault Models: Discussion

IV.1.2 Accident Models

Requirements of an accident model:
- How does an accident happen?
- Which activities can prevent it from happening?

Examples:
- basic energy model
- domino and single event models
- chain of event models
- perturbation theory of accidents
- System theory
- Control theory

Basic Energy Model

Engineering view:
- Accident: result of an uncontrolled and undesired release of energy
- Prevention: altering or controlling the path of energy flow

More detailed considerations:
- Two cases (1) exceed resistance (2) from normal exchange
- An energy transformation accident occurs when transformation of energy injures people or damages property and an energy deficiency accident happens when the required energy is missing

Domino Model

Causality requires a sequence of five steps (dominos):
1. Ancestry or social environment, leading to
2. Fault of person, which is the proximate reason for
3. An unsafe act or condition (mechanical or physical hazard), which results in
4. An accident, which leads to
5. An injury
- Accident can be prevented at any step

Invalid assumption: the accident is always the result of a single event!
Chain of Events Model

- Sequence of events (not single event)
- Problem: where to stop when looking backwards?
- Subjective choice of what should be included
- AND/OR combination (see fault trees)

Accident prevention might include multiple actions

The timing of events is not considered

Multilinear Event Sequence Model

- Multilinear Event Sequence Model:
  - Accident is sequence of events, where an event is defined as one actor plus one action
  - Explains accidents in terms of specific interacting actors

Multilinear Event Sequence Model (2/2)

Perturbation theory of accidents:
- Perturbations = external influences that vary from what is usual or expected
- Stable state as long as actors adapt to the perturbations without being stressed beyond their capacity to adapt or recover. The resulting accident sequence is:
  - If one of the actors fails or is unable to adopt, the perturbation initiates an accident sequence
  - This may over-stress other actors, causing injury or damage, which in turn over-stress subsequently exposed actors
- Cascading injury or damage are possible until the actors are able to accommodate the stresses without harm

Accident Models & System Theory

- Models based on system theory consider accidents as arising from the interaction between components of the system
- Safety is an emergent property that arises when the components which interact within an environment as restricted by its safety constraints
- Accident can result from interactions when one component violates these constraints
- Such an interaction of components leading to a hazard is named dysfunctional interaction
  - Software can contribute to an accident (e.g., controller)

Two types of dysfunctional interaction:
- Problems in articulating and coordinating subsystems
  - Boundary, overlapping control, asynchronous evolution of subsystems
- Lack of linkage
  - E.g., mental model mismatch (user <-> system)

Accident Models & Control Theory

- Safety can be seen control problem
- Safety = stability or system remains in safe region
- Accident equals a disturbance which is not adequately handled by the control system
- Safety is not directly measurable and therefore must be interfered by a systematic and analytic prediction
- Focus on change rather than energy flow
  - Incomplete understanding possible

Accident Models: Summary

Examples:
- Basic energy model
- Domino model
- Chain of events model
- Perturbation theory of accidents
- System theory
- Control theory

Choice of accident model:
- Restrict ways how hazards can be excluded
- Limits the ways hazards may be produced
IV.1.3 Formal Methods

- **Formal methods** are the use of mathematical techniques in the specification, design and analysis of computer hardware and software.
- CASE tools and graphic or diagrammatic methods to describe the requirements or specification of a system are **formalized methods**, only.

Formal methods are based on:
- Formal specifications and formal models
- Techniques to prove equivalence or refinement

Exploit Formal Models

**General observation:**
- Mathematical modelling of the (continuous) system behaviour is at the heart of all forms of control engineering.
- Computer hardware and software works in contrast has a discrete (digital) system behaviour and therefore discrete mathematics and mathematical logic has to be employed.
- The term formal methods covers a wide range of techniques that use discrete mathematics and mathematical logic.

Levels of Rigour (1/2)

Four levels of rigour ([Rushby1993]):
- **Level 0**: No use of formal methods
- **Level 1**: Use of concepts and notations from discrete mathematics
- **Level 2**: Use of formalized specification languages with some mechanized support tools
- **Level 3**: Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking

Levels of Rigour (2/2)

<table>
<thead>
<tr>
<th></th>
<th>Documents</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level 0</strong></td>
<td>Natural language, pseudo code, augmented with diagrams and equations</td>
<td>Manual process of review and inspection</td>
</tr>
<tr>
<td><strong>Level 1</strong></td>
<td>Logic and discrete mathematics is used to replace some natural language parts (less ambiguous)</td>
<td>Only informal proofs if any</td>
</tr>
<tr>
<td><strong>Level 2</strong></td>
<td>Specification languages provide a standardized notation for discrete mathematics which usually provide some automated support for checking for certain class of faults</td>
<td>Only rigorous proofs, but often still not formal proofs</td>
</tr>
<tr>
<td><strong>Level 3</strong></td>
<td>Specification language that employs a strictly defined logic and provides techniques for formal proofs</td>
<td>Only formal proofs which can be developed using theorem provers and checked using proof checkers</td>
</tr>
</tbody>
</table>

Extent of the Application

- Formal methods can be employed for all, some or none of the various verification activities between the process stages.
- Formal methods may be applied for the complete system or only selected components.
- Only certain characteristics (e.g., those related to safety) may be verified using formal methods.

Observation:
- Different formal methods may be employed.
- Translation between them is often "impossible".

Areas of Application

- Requirements analysis
  - Limited value, because good communication with the customers is most important.
- Specification
  - Use of formal specification languages.
- Design and Implementation
  - Proof correct realization step.
  - Stepwise refinement.
- Verification & Validation
  - Use tools to verify specific properties.
Formal Specification Languages (1/2)

Algebraic specification languages (level 2)
- OBJ
  - Executable specification language
  - Behaviour is expressed by rules (equations)
Model-based specification languages (level 2)
- Vienna Development Method (VDM)
  - Operations, states
  - Pre- and post-conditions
  - Programming constructs: while, if...then...else, ...
  - No time concept, but extensions to cope with time possible
  - Only sequential systems

Formal Specification Languages (2/2)

- Z (University of Oxford) (level 2)
  - Based on typed set theory
  - Variables and axioms
  - No time concept, but extensions to cope with time possible
  - Only sequential systems
- Formal methods for concurrent systems (level 2)
  - Temporal logics (CTL, LTL)
  - Process algebras (LOTOS, CCS)
  - Level 3 approaches:
    - Higher Order Logic (HOL)
    - PVS
    - Boyer-Moore

Integration of process algebras in HOL/PVS

Formal Specification Languages (3/3)

Advantages:
- Automatic consistency checks
- Rigorous type checking
- Executable specifications (OBJ, subset of VDM)
- Validation of a trusted unambiguous system description
- Formal proofs for the design and implementation are possible

Disadvantages:
- More specification efforts
- Well trained experts are required
- Detection of subtle errors is hard due to the larger mental distance

Design and Implementation

A given specification is transformed into
- An architectural design
- A detailed design
- An implementation

The models become more detailed with each step
- Often structural similarities with specification
- But also structural difference (e.g., fault tolerance)

Steps required:
- **Realize** the more detailed model
- **Proof** that the model fulfills the model specification

Stepwise Refinement (1/2)

- A series of correct model-transformations (stepwise refinement) can be used to **realize** the required system
- Techniques that assist the required **proof** are often provided (less effective for significant structural changes)

Stepwise Refinement (2/2)

**Stepwise refinement**: A software development technique in which data and processing steps are defined broadly at first and then further defined with increasing detail.

[IEEE-610.12-1990]

Observation:
- In line with most development techniques
- But, in practice changes in later stages will enforce to also change earlier stages to be correct!
Automatic Checking

- Instead of a "complete" specification use only one that consists of relevant properties (e.g., for safety)
- Usually only restricted notions for formal models
  - Finite automata (or similar restricted models)
  - Temporal logic (Model-Checking)
- Often restricted notions for formal properties
  - Prepositional logic
  - Temporal logic (Model-Checking)

Benefits:
- Counterexample when a property is not fulfilled
- For too large models not feasible (state explosion)
- For too complex formal properties not feasible

Industrial Applications (1/2)

- Traffic Alert and Collision Avoidance System (TCAS)
  - After flaws have been found in the original standard formal methods have been used to produce a new formal specification
- Central control function display information system (CDIS)
  - Requirements: world model of the system and its environment (ER), processing requirements (real-time Youdan technique), core specification of all handled data (VDM)
  - Reduced train separation for the Paris Métro (SACEM)
  - A specification written in the B language and reverse engineering to validate the operational code

Industrial Applications (2/2)

- T800 Transputer floating-point unit (INMOS)
  - The work with Z was that successful that even uncovered faults in the IEEE floating-point standard gave been detected
- AAMPS Microprocessor (Rockwell)
  - The operations and the related microcode have been specified using PVS at the instruction set and register transfer level to proof that the microcode correctly implements the operations
- Several railway, avionics and nuclear applications (ABB)
  - Prepositional logic and the Prover tool have been used for these projects and verification time has been reduced by 90% and the overall development time was cut by 20%

Formal Methods: Summary

- Flight warning computer A330/340 (Airbus)
  - A second implementation with formal specification written in LOTOS took about 30% longer, because the design was too large for the tools and the generated code was to slow

Current Status:
- Formal methods are at most used for parts of the development
- Machine supported formal verification remains specialized activity
- The use of formal specification languages is now widespread for safety-critical systems

Formal Methods: Summary

- Also less beneficial cases:
  - Environment assumptions correct?

Summary

Inherent limitation of all (formal) models:
- Effects not covered by the model (abstraction) cannot be detected when considering the model only (validation is always required)
- Models are often only useful, when abstraction is used to hide tedious details not relevant for the problem at hand (building adequate models is crucial)
IV.2 Fault Tolerance

- To ensure safety, the system’s design must “deal” with all anticipated faults
- One strategy to do this are execution-time techniques that cope with the effects of faults and reduce its effects to an acceptable level!

Fault tolerance (FT): Providing a service that is consistent with its specification in spite of faults.

Limitations of Fault Tolerance

Can work only for anticipated faults:
- You can’t tolerate what you don’t expect
- But if we expected it, we would avoid or eliminate the fault!
  ⇒ employ only for faults you cannot avoid/eliminate

In general:
- We can itemize the classes of faults that can occur
- If the fault occurs (the error is detected) we can react on this

Four Phases of Fault Tolerance

(1) Error detection:
- You must know there is a problem in order to deal with it

(2) Damage assessment:
- You must know or at least estimate the damage so as to know how bad the situation is

(3) State restoration:
- A consistent state is needed to continue

(4) Continued service:
- Do something useful with what is left

(1) Error Detection Techniques

- Functionality checking
  Only hardware: e.g., memory checks via checksums

- Consistency checking
  e.g., range checks

- Signal comparison
  Checking pairs

- Information redundancy
  Parity checking, checksums, ...

- Instruction monitoring
  CPU reaction when an invalid instruction code is detected

- Loopback testing
  Feedback output to compare it with source

- Watchdog timers
  reset CPU when a timer is not incremented

- Bus monitoring
  check address ranges on the bus

- Power supply monitoring
  Power supply monitor initiates emergency action before voltage reaches dangerous level

- Uninterruptible power source
  when no disruption can be permitted

(2) Damage Assessment

- How much damage to the system occurs when a component fails? It might be a lot.....

Example: an Ada exception (Ariane 5 accident)

- Failure semantics = defines which divergent behaviour is possible if faults are present
- Components that are expected to fail must have predefined failure semantics
- Hazard analysis reveals which hazardous behaviour of the system might result
- If critical hazards can result, the design has to take care to exclude the involved chain of events

(3) State Restoration / Error Recovery

- There are two possibilities to transform a currently erroneous system state into an error-free system state:
  
  Backward recovery:  
  - system state is reset to a previously store error-free system state  
  - Re-execution of failed processing sequence  
    e.g., database systems (predict valid system states is not possible)
  
  Forward recovery:  
  - system state is set to a new error-free system state  
    typical for real-time systems with periodic processing patterns  
    (it is possible to predict valid system states)
(4) Continued Service

- Some kind of redundancy is required to tolerate faults, because whether or not an error actually leads to a failure depends on the following facts:
  - the system composition and the existence of redundancy (intentional or unintentional redundancy)
  - the system activity after the introduction of an error (the error may get overwritten)
  - the definition of the correct operation (which implicitly defines what is a failure or not)

IV.2.1 Fault-Tolerance and Redundancy

Redundancy can occur in 3 different domains.

(1) Domain of information:
- redundant information e.g. error correcting codes, robust data structures

(2) Domain of space:
- replication of components, e.g. 2 CPU's, UPS (uninterruptible power supply)

(3) Domain of time:
- replication of computations, e.g. calculate results by same (or different) algorithm a second time, sending messages more than once

FT in the Domain of Information

- error correcting codes: for all error correcting codes (ECC)
  \[(2t + p + 1) = d\]
  \[d\] Hamming distance of code
  \[t\] number of single bit errors to be tolerated
  \[p\] number of additional detected errors

- robust data structures:
  - store the number of elements
  - redundant pointers
  - (e.g. double linked chains with status)
  - status or type information
  - checksum or CRC
  - application specific knowledge

FT in the Domain of Space

- Active redundancy
  - parallel fail-silent components
  - voting, triple modular redundancy (TMR)

- Passive or standby redundancy
  - hot standby: standby component is operating in background
  - cold standby: standby components starts only when required

FT in the Domain of Time

- allows tolerance of temporary faults multiple calculation:
  - a function is calculated \(n\) times with the same inputs
  - the result is checked by an acceptance test
  - or the multiple results are voted
  - sending messages multiple times:
    - message transmission is repeated \(n\) times
    - retransmission only in case of failures (positive acknowledge retransmit PAR)
    - retransmission always \(n\) times (reduces temporal uncertainty for real-time systems)

Redundancy and Diversity

- Redundancy with identical components protects against random hardware component failures, but not systematic ones (common mode failures)
  - diversity is also required

- Hardware diversity: micro-controller and hard wired or programmable logic controller (PLC)
- Software diversity: Common mode failures can always result form the specification
IV.2.2 Techniques for FT

There are two fundamental approaches to fault-tolerance:

- **Systematic fault-tolerance**
  - replication of components
  - divergence of components is used for fault-detection
  - redundant components are used for continued service

- **Application-specific fault-tolerance**
  - reasonableness checks for fault detection (based on model of real world)
  - state estimations for continued service

**Application-specific Fault-Tolerance (1/2)**

- the computer system interacts with some physical process, the behaviour of the process is constrained by the law of physics
- these laws are implemented by the computer system to check its state for reasonableness
- for example:
  - the acceleration/deceleration rate of an engine is constrained by the mass and the momentum that affects the axle
  - signal range checks for analogue input signals
  - reasonableness checks are based on application knowledge
  - fail-stop behaviour can be implemented based on reasonableness checks

**Application-specific Fault-Tolerance (2/2)**

- the laws of physics constraining the process can be used to perform state estimations in case some component has failed
- for example:
  - if the engine temperature sensor fails a simple state estimation could assume a default value
  - a better state estimation can be based on the ambient temperature of the engine, engine load and thermostatical behavior of the engine
  - the speed of a vehicle can be estimated if the engine speed and the transmission ratio is known
- state estimations are based on application knowledge
- fail-operational behaviour can be implemented based on reasonableness checks and state estimations

**Systematic Fault-Tolerance (1/2)**

- does not use application knowledge, makes no assumptions on the physical process or controlled object
- uses replicated components instead
- if among a set of replicated components, some — but not all — fail then there will be divergence among replicas
- information on divergence is used for fault detection
- replicas are therefore required to deliver corresponding results in the absence of faults
- The problem of **replica determinism**: due to the limited accuracy of any sensor that maps continuous quantities onto computer representable discrete numbers it is impossible to avoid non-deterministic behaviour

**Systematic Fault-Tolerance (2/2)**

- systematic fault-tolerance requires agreement protocols due to replica non-determinism
- the agreement protocol has to guarantee that correct replicas return corresponding results (the problem of replica determinism is discussed later)
- fail-stop behaviour can be implemented by using the information of divergent results
- fail-operational behaviour can be implemented by using redundant components

**Comparison of Techniques (1/3)**

<table>
<thead>
<tr>
<th>Systematic fault-tolerance</th>
<th>Application-specific fault-tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>replication of components</td>
<td>no replication necessary</td>
</tr>
<tr>
<td>Divergence among replicas in case of faults</td>
<td>—</td>
</tr>
<tr>
<td>no-reasonableness checks necessary</td>
<td>reasonableness checks for fault detection</td>
</tr>
<tr>
<td>requires replica determinism</td>
<td>—</td>
</tr>
<tr>
<td>no application knowledge necessary</td>
<td>depends on application knowledge</td>
</tr>
<tr>
<td>exact distinction between correct and faulty behaviour</td>
<td>fault detection is limited by a grey zone</td>
</tr>
</tbody>
</table>
Comparison of Techniques (2/3)

<table>
<thead>
<tr>
<th>Systematic fault tolerance</th>
<th>Application-specific fault tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>no state estimations necessary</td>
<td>state estimations for continued service</td>
</tr>
<tr>
<td>independence of application areas</td>
<td>missing or insufficient reasonableness checks for some application areas</td>
</tr>
<tr>
<td>service quality is independent of whether replicated components are faulty or not</td>
<td>quality of state estimations is lower than quality derived during normal operation</td>
</tr>
<tr>
<td>correct system function depends on the number of correct replicas and their failure semantics</td>
<td>correct system function depends on the severity of faults and on the capability of reasonableness checks and state estimations</td>
</tr>
<tr>
<td>only backward recovery</td>
<td>forward and backward recovery</td>
</tr>
</tbody>
</table>

Comparison of Techniques (3/3)

<table>
<thead>
<tr>
<th>Systematic fault tolerance</th>
<th>Application-specific fault tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>additional costs for replicated components (if no system inherent replication is available)</td>
<td>no additional costs for replicated components</td>
</tr>
<tr>
<td>no increase in application complexity</td>
<td>considerable increase in application complexity</td>
</tr>
<tr>
<td>considerable increase of system level complexity</td>
<td>no increase of system level complexity</td>
</tr>
<tr>
<td>separation of fault-tolerance and application functionality</td>
<td>application and fault-tolerance are closely intertwined</td>
</tr>
<tr>
<td>fault-tolerance can be handled transparently to the application</td>
<td>fault-tolerance is not handled transparently to the application</td>
</tr>
</tbody>
</table>

Systematic and Application-specific FT

- under practical conditions there will be a compromise between systematic and application-specific fault-tolerance
- usually cost, safety and reliability are the determining factors to choose a proper compromise
- software complexity plays an important role:
  - for complex systems software is almost unmanageable without adding fault-tolerance (fault containment regions and software robustness)
  - therefore systematic fault-tolerance should be applied in favor of application-specific fault-tolerance to reduce the software complexity
- systematic fault-tolerance allows to test and to validate the mechanisms independently of the application software (divide and conquer)

IV.2.3 Hardware Fault Tolerance

- Static redundancy
  - Fault masking to prevent error propagation

- Dynamic redundancy
  - Detection of faults plus actions to nullify them

- Hybrid redundancy
  - Fault masking to prevent error propagation
  - Detection of faults and reconfiguration to remove faulty units from the system

Triple Modular Redundancy (TMR)

- Domain of space
- Static
- Signal comparison (voting)

Advantages:
- Protection against random component failures
- High redundancy costs

Disadvantages:
- Voter a single point of failure

Triplicated Voting

- Domain of space
- Static
- Signal comparison (voting)

Advantages:
- Protection against random component and voter failures

Disadvantages:
- Even higher redundancy costs
Multistage TMR Arrangement

- Even more expensive, but also do not mask two failures occurring in two components of one stage

N-Modular Redundancy

Advantages:
- Protection against (N-1)/2 random component failures

Disadvantages:
- Voter a single point of failure
- Very high redundancy costs

Cold Standby Spares

- Redundancy: Passive/standby and dynamic
- Lower costs than redundancy (component twice + fault detector + switch)

Disadvantages:
- No fault masking
- Reconfiguration may cause a momentary disruption of service while standby unit is activated

Hot Standby Spares

- Redundancy: Passive/standby and dynamic
- Lower costs than redundancy (component twice + fault detector + switch)

Disadvantages:
- No fault masking
- Increases power consumption
- Standby unit has the same stress as the active unit

Self-Checking Pair

- Redundancy: Passive/standby and dynamic
- Lower costs than redundancy (component twice + Comparator)

Disadvantages:
- No fault masking
- Increases power consumption
- Standby unit has the same stress as the active unit
- Comparator single point of failure

Self-Checking Pair (Fail-Silent)

- Redundancy: Passive/standby and dynamic
- Lower costs than redundancy (component + Comparator twice)

Disadvantages:
- No fault masking
- Increases power consumption
- Standby unit has the same stress as the active unit
- No output must result in safe state
Parallel Fail-Silent Components

Fail-Silent Component
- Internal error detection unit prevents faulty result from occurring on the output
- passive or standby redundancy
  - hot standby: standby component is operating in background
  - cold standby: standby components start only when required

Fail-Silent Component
- n

Internal error detection unit
- prevents faulty result form occurring on the output

passive or standby redundancy
- n
- hot standby: standby component is operating in background
- cold standby: standby components start only when required

N-Modular Redundancy with Spares

Redundancy
- Domain of space
- Active and hybrid
- Signal comparison (voting)

Advantages:
- Protection against (N-1)/2 random component failures

Disadvantages:
- Voter a single point of failure
- Very high redundancy costs

IV.2.4 Software Fault Tolerance

Two meanings:
- "Tolerance of software faults" (can be addresses by the techniques for hardware fault tolerance using software diversity)
- "Tolerance of faults by the use of software" (includes first case plus effects of the underlying hardware)

Achieve diversity:
- Usually the same requirements (weakness)
- different programmers, contractors?

Hardware vs. Software

- hardware components are more reliable compared to software components
- very mature technology for hardware process validation
- but: "build it in hardware instead" is no solution at all since the problem of design dependability arises because of the system inherent complexity:
  - very complex systems are realized in software because of their complexity
  - software is often used to implement radically new systems
  - higher flexibility of software is often exploited by very short modification cycles

- Outages in % by fatal faults for the Tandem system illustrates the shift from hardware to software (cf. [Gray1985]):

<table>
<thead>
<tr>
<th></th>
<th>1985</th>
<th>1987</th>
<th>1989</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>19%</td>
<td>13%</td>
<td>10%</td>
</tr>
<tr>
<td>Hardware</td>
<td>25%</td>
<td>22%</td>
<td>7%</td>
</tr>
<tr>
<td>Maintenance</td>
<td>9%</td>
<td>12%</td>
<td>15%</td>
</tr>
</tbody>
</table>

Exception Handling (1/3)

- to detect erroneous states of software modules the exception mechanism can be used (software and hardware mechanisms for detection of exceptional states)
- a procedure (method) has to satisfy a pre condition before delivering its intended service which has to satisfy post conditions afterwards
- the state domain for a procedure can be subdivided:
  - Anticipated exceptional domain
  - Unanticipated exceptional domain
  - Standard domain

Example:
- Ada...
Exception Handling (3/3)

Advantages:
- no voting required
- fault detection distributed in the code (easier)

Disadvantages:
- fault detection distributed in the code (structure?)
- correct run-time handling of exceptions required

Remark:
- complex control structures (difficult verification)

Recovery Blocks (1/4)

- a method to apply diverse designs to provide design fault-tolerance based on an acceptance test—which detects erroneous states—different modules are tried until an acceptable state is reached
- Examples for Acceptance tests:
  - Checks for run-time errors
  - Checks for reasonability
  - Excessive execution time
  - Mathematical errors
  - acceptance tests are application-specific, they have only limited error detection coverage

Recovery Blocks (2/4)

Program scheme:
```
primary module
acceptance test
secondary module
acceptance test
```
Problem:
- execution of a module might corrupt system state
- Recovery Point:
  - backward error recovery
  - use entire system state is inefficient

Idea:
- add recovery point before primary module execution

Program scheme:
```
Establish recovery point
primary module
acceptance test
alternative module 1
acceptance test
alternative module 2
acceptance test ...
```

Recovery Blocks (3/4)

Advantages:
- no voting required
- can also handle (transient) hardware faults
- can be used to implement graceful degradation, when different modules provide different levels of service

Disadvantages:
- additional acceptance test required
- delay for backward recovery in real-time systems
- it is difficult to development acceptance tests
- the quality of acceptance test is often questionable

Recovery Blocks (4/4)

Remarks:
- mixture of systematic and application-specific fault-tolerance:
  - systematic method to apply n diverse modules by rollback recovery
  - acceptance test is application specific
  - recovery blocks can be nested such that a module itself is a recovery block
  - can also be supported with the exception mechanism (e.g. standard exception handler for unidentified exceptions can be used)
  - modeling of recovery block with primary and one alternative is equivalent to passive redundant system (acceptance test ? switch)

Distributed Recovery Blocks

- for uniform treatment of software and hardware failures
- the primary module is executed on the primary processor, the alternate is executed on a backup processor
- both processors use duplicated acceptance test
- if the primary module fails, a message is sent to the backup and the backup then forwards its results
- combination of software and hardware diversity
N-Version Programming (1/4)

- n non-identical replicated software modules are applied and instead of an acceptance test a voter takes a m out of n or majority decision
- driver program to invoke different modules (different processes for module execution), wait for results and voting require more resources than recovery blocks but less temporal uncertainty (response time of slowest module)

Redundancy
- Domain of space or time
- Signal comparison (voting)

Example:
- Primary flight control of the Airbus A330/A340

N-Version Programming (2/4)

Advantages:
- For N=2 like self-checking pair (repeating the execution helps for transient faults; diagnosis to determine faults routine)
- Protection against (N-1)/2 faulty program versions

Disadvantages:
- High implementation costs (>N due to the voting)
- Performance costs of n executions and voting
- Common mode faults are not excluded

N-Version Programming (3/4)

Problems of replica non-determinism:
- the real-world abstraction limitation is no problem (all modules get exactly the same inputs from driver program)
- consistent comparison problem: diverse implementations, different compilers, differences in floating point arithmetic, multiple correct solutions (n roots of nth order equation), ...

What can be done?
- there is no systematic solution for the consistent comparison problem
- either very detailed specification with many agreement points (limits diversity)
- or approximate voting to consider non-determinism (application-specific)

N-Version Programming (4/4)

n-version programming is approach to systematic fault-tolerance:
- there is no application specific acceptance test necessary
- exact voting on every bit is systematic
- modeling of n-version programming is equivalent to active redundant systems with voting

Remark:
- costs make N>2 very uncommon
- Only highly safety-critical systems

Deadline Mechanism

- based on recovery blocks, but deadline instead of acceptance test
- used to avoid timing failures in real-time systems

```plaintext
service name
within response-period
by primary_module
else
by alternate_module
```

- it is assumed that an upper execution bound for the alternate is known
- for the primary it is assumed that the execution is timely in most cases
- if the primary does not finish within the slack time (response-period – execution bound for alternate) then the primary is aborted and the alternate is used

N self-checking programming

- n versions are executed in parallel (similar to N-version programming)
- each module is self-checking, an acceptance test is used (similar to recovery blocks)
- mixture of application specific and systematic fault-tolerance
- requires no backward recovery and no voting
Data Diversity
- It is assumed that software fails on some “special” inputs.
- If the inputs are changed slightly, the same software may work correctly.
- Data re-expression is necessary to generate different but logically equivalent data sets (application specific).
- For real variables, the value may be changed slightly.
- Coordinate transformation to new origin.
- Cheaper alternative to diverse software.

Independence Assumption
- Empirical studies have shown that diverse designed software does not fail independently (co-dependent failures).
  - 27 program versions have been written by two universities.
  - Failure probabilities for 1.15·10^-6 test cases with 301 2-version systems and 2925 3-version systems were calculated.
  - For the average 3-version system, the failure probability improved by a factor 19, compared to the average single version.
- If the independence assumption would hold, the failure probability should have decreased by at least three orders of magnitude.

Problems Due to Co-Dependence
- Software fault-tolerance is based on the independence assumption that predicts that diverse designed models fail independently.
  - Different programmer teams.
  - Different programming language and tools, …
- But …
  - Only modest increase for very high effort.
  - Development costs are main costs for software.
  - Replica non-determinism or application-specific methodology.
  - Increasing costs and time for handling problems for multiple version systems (project management, configuration control, versioning, modifications and updates).

Forms of Redundancy
Control redundancy includes:
- Exception handling.
- Recovery blocks.
- N-version programming.
- Self-checking programming.
- Deadline mechanism.
- Data diversity.

Data redundancy uses extra data:
- To check the validity of results.
- Error correcting/detecting codes.
- Checksum agreements, etc.

Summary
- N-version programming similar to N-modular redundancy.
- Recovery blocks similar to dynamic redundancy.
- Duplicated identical hardware modules provide fault tolerance for some form of hardware faults whereas duplication of identical software has little benefit (only transient faults).
- Software redundancy required diversity (due to the high costs usually preserved for highly critical applications).

IV.2.5. Replication
- Problems
  - The Problems of Replica Determinism.
  - Non-deterministic behaviour.
  - Limits of Redundancy.
- Replica control
  - Internal vs. external.
  - Centralized vs. distributed.
  - Control strategies.
  - Failure recovery.
  - Redundancy preservation.
- Complexity.
The Problem of Replica Determinism

- For systematic fault-tolerance it is necessary that replicated components show consistent or deterministic behaviour in the absence of faults.
- If for example two active redundant components are working in parallel, both have to deliver corresponding results at corresponding points in time.
- This requirement is fundamental to differentiate between correct and faulty behaviour.
- At a first glance it seems trivial to fulfil replica determinism since computer systems are assumed to be examples of deterministic behaviour, but in the following it is shown that computer systems behave almost deterministically.

Non-deterministic Behaviour (1/6)

- **Inconsistent inputs:** If inconsistent input values are presented to the replicas then the results may be inconsistent too.
  - A typical example is the reading of replicated analogue sensors:
    - $\text{read}(C1) = 99.99 \, ^\circ\text{C}$, $\text{read}(C2) = 100.00 \, ^\circ\text{C}$
  - **Inconsistent order:** If service requests are presented to replicas in different order then the results will be inconsistent.

Non-deterministic Behaviour (2/6)

- **Inconsistent membership information:** Replicas may fail or leave groups voluntarily or new replicas may join a group. If replicas have inconsistent views about group membership it may happen that the results of individual replicas will differ.

Non-deterministic Behaviour (3/6)

- **Non-deterministic program constructs:** Besides intentional non-determinism, like random number generators, some programming languages have non-deterministic program constructs for communication and synchronization (Ada, OCCAM, ...).
  - Ada example:
    ```
    task server is
    entry service_1();
    ...
    entry service_n();
    end server;
    ```

Non-deterministic Behaviour (4/6)

- **Local information:** If decisions with a replica are based on local knowledge (information which is not available to other replicas) then the replicas will return different results.
  - System or CPU load
  - Local time
- **Timeouts:** Due to minimal processing speed differences or due to slight clock drifts it may happen that some replicas locally decide to timeout while others do not.
- **Dynamic scheduling decisions:** Dynamic scheduling decides in which order a series of service requests are executed on one or more processors. This may cause inconsistent order due to:
  - Non-identical sets of service requests
  - Minimal processing speed differences

Non-deterministic Behaviour (5/6)

- **Message transmission delays:** Variability in the message transmission delays can lead to different message arrival orders at different servers (for point-to-point communication topologies or topologies with routing).
Non-deterministic Behaviour (6/6)

The consistent comparison problem:
- computers can only represent finite sets of numbers
- it is therefore impossible to represent the real numbers exactly they are rather approximated by equivalency classes
- if the results of arithmetic calculations are very close to the border of equivalency classes, different implementations can return diverging results.
- different implementations are caused by: N-version programming, different hardware, different floating point libraries, different compilers
- for example the calculation of \( (a - b)^2 \) with floating point representation with a mantissa of 4 decimal digits and rounding

\[
(a - b)^2 = 1.000 104 \quad (a - b)^2 = a^2 - 2ab + b^2 = 9.999 103
\]

Limitations to Replication (1/2)

The real world abstraction limitation:
- dependable computer systems usually interface with continuous real-world quantities:
  - quantity SI-unit
  - distance meter [m]
  - mass kilogram [kg]
  - time second [s]
  - electrical current ampere [A]
  - thermodynamic temperature degree kelvin [K]
  - gram-mole mole [mol]
  - luminous intensity candela [cd]
- these continuous quantities have to be abstracted (or represented) by finite sets of discrete numbers
- due to the finite accuracy of any interface device, different discrete representations will be selected by different replicas

Limitations to Replication (2/2)

The impossibility of exact agreement:
- due to the real world abstraction limitation it is impossible to avoid the introduction of replica non-determinism at the interface level
- but it is also impossible to avoid the once introduced replica non-determinism by agreement protocols completely
- exact agreement would require ideal simultaneous actions, but in the best case actions can be only simultaneous within a time interval
- Intention and missing coordination:
  - replica non-determinism can be introduced intentionally
  - or unintentionally by omitting some necessary coordinating actions

Internal vs. External Replica Control

Internal replica control:
- avoid non-deterministic program constructs, uncoordinated timeouts, dynamic scheduling decisions, diverse program implementations, local information, and uncoordinated time services
- can only be enforced partially due to the fundamental limitations to replication

External replica control:
- control non-determinism of sensor inputs
- avoid non-determinism introduced by the communication service
- control non-determinism introduced by the program execution on the replicated processors by exchanging information

Groups and Replication Level

Replicated entities such as processors are called groups.
- The number of replicas in a group is called replication level
- A group is said to be \( n \)-resilient if up to \( n \) processor failures can be tolerated

Group vs. hierarchical failure masking
- Group failure masking: The group output is a function of the individual group members output (e.g. a majority vote, a consensus decision). Thus failures of group members are hidden from the service user.
- Hierarchical failure masking: The processors within a group come up with diverging results and the faults are resolved by the service user one hierarchical level higher.
Basic Services for Groups

The basic services for replicated fault-tolerant systems

- Membership: Every non-faulty processor within a group has timely and consistent information on the set of functioning processors which constitute the group.
- Agreement: Every non-faulty processor in a group receives the same service requests within a given time interval.
- Order: Explicit service requests as well as implicit service requests, which are introduced by the passage of time, are processed by non-faulty processors of a group in the same order.

Central vs. Distributed Replica Control

- Strictly central replica control:
  - there is one distinguished processor within a group called leader or central processor
  - the leader takes all non-deterministic decisions
  - the remaining processors in the group, called followers, take over the leaders decisions

- Strictly distributed replica control:
  - there is no leader role, each processor in the group performs exactly the same way
  - to guarantee replica determinism the group members have to carry out a consensus protocol on non-deterministic decisions

Replica Control Strategies (1/4)

Lock-step execution:
- processors are executing in synchronous
- the outputs of processors are compared after each single operation
- typically implemented at the hardware level with identical processors

Advantages:
- arbitrary software can be used without modifications for fault-tolerance (important for commercial systems)
Disadvantages:
- common clock is single point of failure
- transient faults can affect all processors at the same point in the computation
- high clock speed limits number and distance of processors
- restricted failure semantics

Replica Control Strategies (2/4)

Active replication:
- all processors in the group are carrying out the same service requests in parallel
- strictly distributed approach, non-deterministic decisions need to be resolved by means of an agreement protocol
- the communication media is the only shared resource

Advantages:
- unrestricted failure semantics
- no single point of failure
Disadvantages:
- requires the highest degree of replica control
- high communication effort for consensus protocols
- problems with dynamic scheduling decisions and timeouts

Replica Control Strategies (3/4)

Semi-active replication:
- intermediate approach between distributed and centralized
- the leader takes all non-deterministic decisions
- the followers are executing in parallel until a potential non-deterministic decision point is reached

Advantages:
- no need to carry out a consensus protocol
- lower complexity of the communication protocol (compared to active replication)
Disadvantages:
- restricted failure semantics, the leaders decisions are single points of failures
- problems with dynamic scheduling decisions and timeouts

Replica Control Strategies (4/4)

Passive replication:
- only one processor in the group – called primary – is active
- the other processors in the group are in standby
- checkpointing to store last correct service state and pending service requests

Advantages:
- requires the least processing resources
- standby processors can perform additional tasks
- highest reliability of all strategies (if assumption coverage = 1)
Disadvantages:
- restricted failure semantics (crash or fail-stop)
- long resynchronization delay
Failures and Replication (1/2)

Centralized replication:
- semi-active and passive replication
- the leading processor is required to be fail restrained
- Byzantine or performance failures of the leader cannot be detected by other processors in the group ("heartbeat" or "I am alive" messages)
- to tolerate $t$ failures with crash or omission semantics $t + 1$ processors are necessary
- the result of any processor (e.g. the fastest) can be used
- if no reliable broadcast service is available

Failures and Replication (2/2)

Distributed replication:
- active replication
- no restricted failure semantics of processors
- to tolerate $t$ crash or omission failures $t + 1$ processors are necessary
- to tolerate $t$ performance failures $2t + 1$ processors are necessary
- for crash or omission failures it is sufficient to take 1 processor result
- for performance or Byzantine failure $t + 1$ identical results are required

Failure Recovery (1/2)

After occurrence of a failure (that is covered by the fault hypothesis) the group has to perform some recovery actions

Centralized replication:
- failures of followers require no recovery actions
- if a leader fails a new leader needs to be elected
- then the new leader has to take over the service of the failed leader
- typically solved by backward recovery (reexecution from last fault free state)
- recovery time needs to be considered for real-time services
- window of vulnerability where new leader cannot decide whether the last output was made successfully or not
- output devices typically require at least once semantics (state semantics)

Failure Recovery (2/2)

Distributed replication:
- no special recovery actions necessary since all services are executed in parallel
- no election in case of processor failures
- output devices have to consider the results of all group members or each group member has its own output device (idempotence)
- no state semantics for output devices necessary (exactly once semantics possible)

Redundancy Preservation (1/2)

to guarantee fault-tolerance and to cover the fault hypothesis the replication level has to be kept above a given threshold

assuming $n$ processors are in a group where $f$ have failed and up to $t$ failures have to be tolerated then one of the following combining conditions needs to be satisfied:
- $n - f > 2t$ for Byzantine failures
- $n - f > t$ for performance failures
- $n - f > 0$ for crash or omission failures

if this combining condition is violated
- a new processor needs to be added to the group (redundancy preservation)
- or the service of the group has to be abandoned

Redundancy Preservation (2/2)

real-time requirements for redundancy preservation need special consideration
- faults in the reconfiguration service need to be considered
- $f$ is therefore the number of failed processors plus the number of correct processors that are configured by a faulty reconfiguration service
- this requires a membership protocol:
  - detect departures and joins of processors to groups
  - provide consistent and timely group membership information on a system wide basis
- joins of processors are difficult to handle:
  - the new processor needs to be synchronized to the service state of the group
  - but the groups service state is evolving over time
  - after synchronization it has to be guaranteed that all further service requests are delivered to the new group member as well
Failure Coverage vs. Complexity

- High assumption coverage implies high complexity
  - for Byzantine faults the assumption coverage is 1
  - Byzantine faults require consensus protocols and very complex
    fault-tolerance mechanisms
  - high probability of faults in the fault-tolerance mechanisms (35% ESS-1)
  - due to the high complexity the system will have a low dependability
- Low assumption coverage implies low dependability
  - low assumption coverage implies high possibility of assumption
    violations
  - in case of assumption violations a fault-tolerant system can fail
    completely
  - the system will therefore have a low dependability
- for optimal dependability a compromise between the assumption coverage rate and complexity of the fault-tolerance mechanism has to be made

IV.2.6 Recovery

- systematic fault-tolerance is often based on backward recovery to recover a consistent state
- in distributed systems a state is said to be consistent if it could exist in an execution of the system
- Recovery line: A set of recovery points form a consistent state—called recovery line—if they satisfy the following conditions:
  1. the set contains exactly one recovery point for each process
  2. No orphan messages: There is no receive event for a message \( m \) before process \( P_i \)'s recovery point which has not been sent before process \( P_j \)'s recovery point.
  3. No lost messages: There is no sending event for a message \( m \) before process \( P_i \)'s recovery point which has not been received before process \( P_j \)'s recovery point.

The Domino Effect

- the consistency requirement for recovery lines can cause a flurry of rollbacks to recovery points in the past
- to avoid the domino effect:
  - coordination among individual processors for checkpoint establishment
  - restricted communication between processors

Synchronous Checkpointing

- based on synchronized clocks check points are established with a fixed period \( p \) by all processes, where \( \beta \) is the clock synchronization precision and \( d \) temporal uncertainty of message transmission
- if a message is sent during \( [T - \beta - d, T] \) it will be received before \( T + \beta + d \)
- to achieve a consistent state two possibilities exist:
  - prohibit message sending during interval \( \beta \) after checkpoint establishment
  - establish checkpoint earlier, at \( kp - \beta - d \) and log messages during the critical instant

Stable Storage (1/2)

- stable storage is an important building block for many operations in fault-tolerant systems (fail-stop systems, dependable transaction processing, ...)
- there are two operations which should work correctly despite of faults (as covered by the fault hypothesis):
  - procedure writeStableStorage(address, data)
  - procedure readStableStorage(address ) returns (status, data)
- many failures can be handled by coding (CRC’s) but other types cannot be handled by this technique:
  - Transient failures: The disk behaves unpredictably for a short period of time.
  - Bad sector: A page becomes corrupted, and the data stored cannot be read.
  - Controller failure: The disk controller fails.
  - Disk failure: The entire disk becomes unreadable.

Stable Storage (2/2)

- Disk shadowing
  - a set of identical disk images is maintained on separate disks
  - in case of two disks this technique is called disk mirroring for performance and availability reasons the disks should be “dual-ported” (e.g. Tandem system)
- Redundant Array of Inexpensive Disks (RAID)
  - data is spread over multiple disks by “bit-interleave” (individual bits of a data word are stored on different disks)
  - in the following example single bit failures can be tolerated since a parity bit is stored on a check disk and disks are assumed to detect single bit failures
  - RAID’s provide high reliability and I/O throughput (parallel read/write)
Example: Fail Stop Processors

- the visible effects of the failure of a fail stop processor are:
  
  1. It stops executing
  2. The internal state of the processor and the volatile storage connected to the processor are lost; the state of the stable storage is unaffected.
  3. Any processor can detect the failure of a fail stop processor.

- real processors do not have such a simple well defined semantics
- typically fail stop processors are implemented by a group of regular processors
- k-fail-stop processor: A processor is said to be k-fail-stop if it can tolerate up to k component (processor) failures while preserving its fail-stop property.

Fail Stop Processors with Stable Storage

Assumptions:
- the stable storage is reliable
- k-fail-stop processors
- communication is reliable
- message origin can be authenticated (point-to-point or cryptographic check)
- synchronous system model (synchronized clocks, bounded communication)

Implementation:
- requests to the stable storage are only granted if k + 1 requests are received within a time interval d

S-process:

\[ S \text{-process: } \]

\[ R := \text{bag of received requests with proper timestamp} \]

\[ \text{if } (|R| = n+1 \text{ } \Rightarrow \text{ all requests are identical } \Rightarrow \text{ failed}) \]

\[ \text{then} \]

\[ \text{if request is write} \]

\[ \text{writeStableStorage} \]

\[ \text{elseif(request is read)} \]

\[ \text{readStableStorage and send result to all processors} \]

\[ \text{fi} \]

\[ \text{else} \]

\[ k \text{-fail stop processor has failed} \]

\[ \text{writeStableStorage failed} \]

\[ \text{fi} \]

Fail Stop Processors without Stable Storage

Assumptions:
- the storage processor are not reliable and can fail byzantine
- k + 1 p-processors (program processors)
- 2k + 1 s-processors (storage processors)
- each s-process has a copy of the stable storage
- communication is reliable
- message origin can be authenticated (point-to-point or cryptographic check)
- synchronous system model

Implementation:
- requests to the stable storage subsystem are only granted if k + 1 requests are received within a time interval d
- failures of individual storage processors are masked by Byzantine agreement (under the assumption of authentication detectable failures)

IV.2.7 Summary

- Forms of Redundancy
- Techniques for Fault Tolerance
- Hardware Fault Tolerance Techniques
- Software Fault Tolerance Techniques
- Problems of Replication
- Recovery

Remember...

- Fault tolerance is not a system property.
- It is a technique by which dependability might be achieved.

Dependability/safety are the required system property.

IV.3 Reliability Modelling

Design phase:
- Designing and implementing the computer system to achieve the dependability required.
  - avoid construction of costly prototypes
  - modeling is used to evaluate design alternatives
  - analysis of critical components
  - for prediction of dependability parameters (safety, reliability, availability, maintainability, ...)

Validation phase:
- Gaining confidence that a certain dependability goal (requirement) has been attained.
  - modeling is used to evaluate the computer system
  - certification
Modeling Techniques

Deterministic modeling
- the maximum number of faults that can be tolerated without system failure is considered
- the evaluation criteria is n-resiliency, i.e. a system is said to be n-resilient if it can tolerate up to n component failures

Probabilistic (quantitative) modeling
- component failure and repair rates are described as stochastic processes
- consideration of failure rates
- statistical models

Probabilistic Modeling

There are three different forms of information which can be used to model a system’s dependability.

- **Historical information (statistics):** Information about the behavior of identical or similar components in the past is assessed.
- **Experimental information (statistics):** Information is gained by exercising the system or single components (for software this includes typically the test and debugging).
- **Structural information:** Overall dependability of a system is deduced from the structure and the dependability figures of its parts.

Probabilistic Functions

Reliability $R(t)$
- the probability that the system will conform to its specification throughout a period of duration $t$.

Failure Probability $Q(t)$
- the probability that the system will not conform to its specification throughout a period of duration $t$.

**Probability Density & Failure Rate**

*Probability density function (Def.):*
The failure density $f(t)$ at time $t$ is defined by the number of failures during $\Delta t$.

$$f(t) = \frac{dQ(t)}{dt} = -\frac{dR(t)}{dt}$$

*Failure rate (Def.):*
The failure rate $\lambda(t)$ at time $t$ is defined by the number of failures during $\Delta t$ in relation to the number of correct components at time $t$.

$$\lambda(t) = \frac{f(t)}{R(t)} = -\frac{dR(t)}{dt} \cdot \frac{1}{R(t)}$$

**Constant Failure Rate**

Used to model the normal-life period of the bathtub curve

- failure rate
- probability density function $f(t) = \lambda e^{-\lambda t}$
- reliability $R(t) = e^{-\lambda t}$

**Failure Rates**

<table>
<thead>
<tr>
<th>Device/Type</th>
<th>FIT</th>
<th>Device/Type</th>
<th>FIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL-SSI, -MSI</td>
<td>3 Transistor</td>
<td>-</td>
<td>200</td>
</tr>
<tr>
<td>CMOS-SSI</td>
<td>3 Transistor (FT)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>RAM (&lt; 1 MB)</td>
<td>20 Power Transistor</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>RAM (&gt; 1 MB)</td>
<td>45 Diode</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>120 Zero-Coupler</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>EPROM (&lt; 1 MB)</td>
<td>10 LED/CCD Display</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>PROM (&gt; 1 MB)</td>
<td>20 Resistor</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LC</td>
<td>20 Lamp 12V</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>LED</td>
<td>40 Lamp 24V</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>CMOS-DA (10 stages)</td>
<td>20 Comparator</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>CMOS-DA (&gt; 10 stages)</td>
<td>20 Switch dimmer</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>OP-Amp</td>
<td>3 Relais</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>Analog custom design</td>
<td>55 Solder joint</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

FIT = failures / 10 [h]
Weibull Distributed Failure Rate

- Used to model infant mortality and wear out period of components.
- \( a < 1 \): failure rate is decreasing with time
- \( a = 1 \): constant failure rate
- \( a > 1 \): failure rate is increasing with time

\[ \lambda(t) = a \lambda(t)^{a-1} \]

probability density function

\[ f(t) = a \lambda(t)^a \exp(-\lambda(t)^a) \]

reliability

\[ R(t) = \exp(-\lambda(t)^a) \]

Lognormal Distributed Failure Rate

- For semiconductors the lognormal distribution fits more data collections than any other and is assumed to be the proper distribution for semiconductor life.

\[ \lambda(t) = \frac{1}{\sqrt{2\pi \sigma^2}} \exp\left[\frac{-(\ln(t) - \mu)^2}{2\sigma^2}\right] \]

probability density function

\[ f(t) = \frac{1}{\sqrt{2\pi \sigma^2}} \exp\left[\frac{-(\ln(t) - \mu)^2}{2\sigma^2}\right] \]

reliability

\[ R(t) = 1 - \int_{\ln(t)}^{\infty} \frac{1}{\sqrt{2\pi \sigma^2}} \exp\left[\frac{-(\ln(x) - \mu)^2}{2\sigma^2}\right] dx \]

IV.3.1 Probabilistic Structural Based Modeling

Assumptions:
- identifiable (independent) components
- each component is associated with a given failure rate
- model construction is based on the structure of the interconnections between components

Models:
- Simple block diagrams
- Arbitrary block diagrams
- Markov models
- Generalized Stochastic Petri Nets (GSPN)

Simple Block Diagrams

- assumption of independent components
- combination of connected components

A) series

\[ R_{\text{series}}(t) = R_1(t) \cdot R_2(t) \cdot \ldots \cdot R_n(t) \]

B) parallel

\[ R_{\text{parallel}}(t) = 1 - (1 - R_1(t)) \cdot (1 - R_2(t)) \cdot \ldots \cdot (1 - R_n(t)) \]

Constant Failure Rate: Series Connection

Given failure rate and reliability

\[ \lambda(t) = \lambda \]

\[ R(t) = \exp(-\lambda t) \]

the resulting failure rate for the system is still constant

Constant Failure Rate: Parallel Connection

Given failure rate and reliability

\[ \lambda(t) = \lambda \]

for 3 parallel components this gives:

\[ R_{\text{parallel}}(t) = 1 - \prod_{i=1}^{3} (1 - R_i(t)) \]

under the assumption \( \lambda_1 = \lambda_2 = \lambda_3 \) it follows

\[ R_{\text{parallel}}(t) = \exp(-3\lambda t) \]

the resulting failure rate is no longer constant!
Simple Block Diagrams

- can be used to model arbitrary combinations of series and parallel connected components
- easy mathematics for constant failure rates

\[
\begin{array}{c}
R_i(t) = \frac{R(t)}{R(t)} \\
R_j(t) = \frac{R(t)}{R(t)} \\
R_k(t) = \frac{R(t)}{R(t)} \\
R_l(t) = \frac{R(t)}{R(t)} \\
\end{array}
\]

Problems
- assumption of independent failures
- maintenance cannot be modeled
- restricted to series/parallel connection
- only for active redundancy

Analysis with Minimal Cut and Tie Sets

- for general networks without feedback the reliability can be analyzed using minimal cut and tie sets
- For \( N_c \) the number of minimal cuts and \( n_j \) the number of elements j-th cut we have the lower bound for reliability

\[
R(t) > 1 - \prod_{j=1}^{N_c} [1 - R(t^j)]
\]

- For \( N_T \) the number of minimal ties and \( n_j \) the number of elements j-th cut we have the upper bound for reliability

\[
R(t) < \sum_{j=1}^{N_T} \prod_{i=1}^{n_j} R(t)
\]

Problems
- assumption of independent failures
- maintenance cannot be modeled
- restricted to series/parallel connection
- only for active redundancy

Active Redundancy and Voting

For TMR 2 out of 3 components have to function correctly

\[
R_{TMR}(t) = R(t) + 3R(t)Q(t)
\]

under the assumption of identical failure rates

\[
R_{TMR}(t) = R(t)^3 + 3R(t)^2Q(t)
\]

for general voting systems where \( c \) out of \( n \) components have to function correctly

\[
R_{AV}(t) = \sum_{k=c}^{n} \binom{n}{k} R(t)^k (1 - R(t))^{n-k}
\]

Active Redundancy

For TMR 2 out of 3 components we have

\[
R_{TMR}(t) = R(t)^3 + 3R(t)^2Q(t) = 3R(t)^2 - 2R(t)
\]

and thus for \( R(t) \) decreasing from 1 to 0 a \( t_S = 0.5 \) exists with

\[
R_{TMR}(t) = R(t)!
\]

Remarks:
- In a TMR the resulting reliability is only greater then that of the modules if the module reliability is greater than 0.5
- Reliability cannot be produced by combining unreliable modules as long as failure detection is not exploited!

Passive vs. Active Redundancy

\[
R(t) = R(t)^n + n(n-1)R(t)^{n-2} \sum_{i=0}^{n-1} R(t)^i
\]

Neglected issues:
- coverage of fail silence assumption
- reliability of voter

Parallel Fail Silent vs. Majority Voting

n = 1 single component
n = 2 two parallel components
n = (3,2) voting, 2 out of 3
n = (5,2) voting, 2 out of 5
Imperfect Passive Redundancy
- under practical conditions it is impossible to build an ideal passive replicated system
- an unreliable switch with \( p = 0.5 \) or a switch with error detection coverage of 80% reduces the system reliability below that of active redundant components

\[
R(n) = \begin{cases} 
0.5 & n = 1 + 1 \\
0.8 & n = 1 + 2 \\
0.8 & n = 2 + 1 \\
1 & n = 2 + 2 \\
0.5 & n = 3 + 1 \\
0.8 & n = 3 + 2 \\
0.8 & n = 4 + 1 \\
1 & n = 4 + 2 \\
0.5 & n = 5 + 1 \\
0.8 & n = 5 + 2 \\
0.8 & n = 6 + 1 \\
1 & n = 6 + 2 \\
0.5 & n = 7 + 1 \\
0.8 & n = 7 + 2 \\
0.8 & n = 8 + 1 \\
1 & n = 8 + 2 \\
0.5 & n = 9 + 1 \\
0.8 & n = 9 + 2 \\
0.8 & n = 10 + 1 \\
1 & n = 10 + 2 \\
0.5 & n = 11 + 1 \\
0.8 & n = 11 + 2 \\
0.8 & n = 12 + 1 \\
1 & n = 12 + 2 \\
0.5 & n = 13 + 1 \\
0.8 & n = 13 + 2 \\
0.8 & n = 14 + 1 \\
1 & n = 14 + 2 \\
0.5 & n = 15 + 1 \\
0.8 & n = 15 + 2 \\
0.8 & n = 16 + 1 \\
1 & n = 16 + 2 \\
0.5 & n = 17 + 1 \\
0.8 & n = 17 + 2 \\
0.8 & n = 18 + 1 \\
1 & n = 18 + 2 \\
0.5 & n = 19 + 1 \\
0.8 & n = 19 + 2 \\
0.8 & n = 20 + 1 \\
1 & n = 20 + 2 \\
0.5 & n = 21 + 1 \\
0.8 & n = 21 + 2 \\
0.8 & n = 22 + 1 \\
1 & n = 22 + 2 \\
0.5 & n = 23 + 1 \\
0.8 & n = 23 + 2 \\
0.8 & n = 24 + 1 \\
1 & n = 24 + 2 \\
0.5 & n = 25 + 1 \\
0.8 & n = 25 + 2 \\
0.8 & n = 26 + 1 \\
1 & n = 26 + 2 \\
0.5 & n = 27 + 1 \\
0.8 & n = 27 + 2 \\
0.8 & n = 28 + 1 \\
1 & n = 28 + 2 
\end{cases}
\]

Single Parametric Measures
Mean time to failure (MTTF):
\[
MTTF = \frac{1}{\mu(t)} \int_0^{t_m} \mu(t) \, dt
\]
Mean time to repair (MTTR):
\[
MTTR = \int_0^{t_m} \frac{1}{\mu(t)} \, dt
\]
Mean time between failures (MTBF):
\[
MTBF = \frac{1}{\mu(t)} \int_0^{t_m} \mu(t) \, dt
\]
Mission reliability:
\[
R_m = R(t_m) \quad \text{for } t_m \text{ the mission duration}
\]
(Steady state) availability:
\[
A = \frac{MTTF}{MTTF + MTTR}
\]

Mean Time to Failure
For constant failure rate \( \mu(t) \), we have
\[
MTTF = \frac{1}{\mu(t)} \int_0^{t_m} \mu(t) \, dt = \frac{1}{\mu(t)}
\]
Series
\[
MTTF_{\text{series}} = \frac{1}{\sum_{i=1}^{n} \lambda_i}
\]
Parallel with \( \lambda_1 = \ldots = \lambda_n \)
\[
MTTF_{\text{parallel}} = \frac{1}{\lambda_1 + \lambda_2 + \cdots + \lambda_n}
\]
Passive
\[
MTTF_{\text{passive}} = \frac{1}{\sum_{i=1}^{n} \lambda_i}
\]

Mission Reliability
- assumption of a mission time \( t_m \)
- during mission there is no possibility of maintenance or repair
- typical examples are space flights or air planes
- suitability of architectures depends on mission time

Repair Rate & MTTR
- repair rate \( \mu(t) \) analogous to failure rate
- most commonly constant repair rates \( \mu(t) = \mu \)
Mean time to repair
- analogous to mean time to failure
\[
MTTR = \frac{1}{\mu(t)}
\]

Availability
- the percentage of time for which the system will conform to its specification
- also called steady state or instantaneous availability
\[
A = MTTF/MTBF = MTTF/(MTTF + MTTR)
\]
- without maintenance and repair \( (t_m) \):
\[
MTTR = 0 \Rightarrow A = 0
\]
- Mission availability \( A(t_m) \):
\[
A(t_m) = \frac{1}{t_m} \int_0^{t_m} R(t) \, dt
\]
IV.3.2 Markov Models

suitable for modeling of:
- arbitrary structures (active, passive and voting redundancy)
- systems with complex dependencies (assumption of independent failures is no longer necessary)
- coverage effects

Markov property:
- The system behavior at any time instant t is independent of history (except for the last state).
- restriction to constant failure rates

The Phases for Markov Modeling

Model design:
- identification of relevant system states
- identification of transitions between states
- construction of Markov graph with transition rates

Model evaluation:
- Differential equation + solution of equation gives \( R(t) \)
  - explicit (by hand)
  - Laplace transformation
  - numeric solution (tool based)
- Integration of differential equation gives MTTF
  - system of linear equations

Model for Active Redundant System

An example with two parallel connected components A and B with maintenance. The failure rates are \( \lambda_A \) and \( \lambda_B \) and the repair rates are \( \mu_A \) and \( \mu_B \).

Identification of states:
1: A correct B correct \( P_1(t) \)
2: A failed B correct \( P_2(t) \)
3: A correct B failed \( P_3(t) \)
4: A failed B failed \( P_4(t) \)

Effect of Maintenance

repair and failure rate: \( \lambda = 1/1000 \text{ [h]} \) \( \mu = 1/10 \text{ [h]} \)

- for 2 active redundant components the MTTF is improved by a factor 34
- for 2 passive redundant components the MTTF is improved by a factor 51

Failure Semantics and Assumption Coverage

comparing a system with two active replicated components to a TMR system shows that under ideal conditions active replication has a higher reliability

But: active replication is based on the assumption that components are fail silent
- assumption coverage ???

TMR voting is based on the assumption of fail consistent components
- assumption coverage < 1
  (if properly constructed)

Failure Semantics and Assumption Coverage

modeling of the TMR was reasonable since assumption coverage of fail consistent behavior = 1
modeling of the active redundant system was idealistic since assumption coverage of fail silent behavior < 1

Markov model:
- \( \lambda \) failure rate for active redundant parallel connected components
- \( c_\text{a} \) assumption coverage for fail silent behavior

\[ \begin{align*}
\lambda_A & = \lambda_B \\
\mu_A & = \mu_B \\
\lambda & = \lambda_A + \lambda_B \\
\mu & = \mu_A + \mu_B
\end{align*} \]
Failure Semantics and Assumption Coverage
- comparing parallel components to a TMR systems shows that the reliability of the parallel system is superior for reasonable assumption coverages, but what about safety (analytic notion of safety)?

Safety $S(t)$ is the probability that the system will not exhibit a specified undesired behavior throughout a period of duration $t$.

Parallel system: $R(t) = S(t)$
- for the parallel components the system reliability is equal to the system safety since the system may potentially cause a hazard if it does not function correctly.

TMR system: $R(t) < S(t)$
- for TMR systems the reliability is not equal to the safety since the system can be in a safe state although it is not functioning correctly, e.g. all three components disagree (if system reacts accordingly).

Safety of a TMR system
To model the safety of a TMR system it needs to be differentiated between incorrect function and the unsafe system state.

Markov model:
- $\lambda$ ... failure rate for single component
- $c$ ... probability of coincident failures of two components

Effect on Safety
failure rate of a single component: $\lambda = 100$ FIT

<table>
<thead>
<tr>
<th>System Description</th>
<th>MTTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = (2, 3), 10$^{-6}$ TMR system, probability of two coincident failures 10$^{-6}$</td>
<td>333,36 10$^{-6}$</td>
</tr>
<tr>
<td>n = (2, 3), 4$^{-5}$ TMR system, probability of two coincident failures 4$^{-5}$</td>
<td>692,71 10$^{-6}$</td>
</tr>
<tr>
<td>n = (2, 3), 0.5 TMR system, probability of two coincident failures 0.5</td>
<td>13,22 10$^{-6}$</td>
</tr>
<tr>
<td>n = 2, 0.999 two parallel comp., coverage of full silent assumption 99.9%</td>
<td>14,00 10$^{-6}$</td>
</tr>
<tr>
<td>n = 2, 0.99 two parallel comp., coverage of full silent assumption 99%</td>
<td>14,00 10$^{-6}$</td>
</tr>
</tbody>
</table>

**Effect on Safety**

- $10 \times 10^{-6}$ probability that two 16 bit numbers coincide
- $4 \times 10^{-3}$ probability that two 8 bit numbers coincide
- $0.5$ probability that two 1 bit numbers coincide

Generalized Stochastic Petri Nets (GSPN)
- because of the very limited mechanisms available, Markov models become easily very complex.
- Petri Nets provide much richer mechanisms, they can be used to model and analyze arbitrary systems, algorithms and processes.
- basic Petri Nets — which were restricted to discrete behavior only — can be extended to “Generalized Stochastic Petri Nets” by allowing transition delays to be either deterministically equal to zero or exponentially distributed random variables, or to be random variables with different distributions.
- it was shown that bounded stochastic Petri Nets are isomorphic to finite continuous Markov chains, i.e. for each stochastic Petri Net there exists a functional equivalent Markov chain (and vice versa).

Petri Net Example
- Single-writer/multiple-reader access to a shared resource
- p1, p2, ... places
- t1, t2, ... transitions
- p1, p2, ... priorities

the 3 tokens in place p1 represents customers that may request the resource firing t1 starts the protocol
- t2 indicates “read” and t3 “write” access, respectively
- the single token in p5 represents the resource.
GSPN Modeling

- To model and analyze a system by means of GSPN the following steps has to be carried out:
  - model construction: usually by means of structured techniques, bottom-up or top-down
  - model validation: structural analysis, possibly formal proves of some behavioral properties
  - definition of performance indices: definition of markings and transition firings (deterministically or stochastic)
  - conversion to Markov chain: generation of reachability set and reachability graph to obtain the Markov chain
  - solution of the Markov chain: Tool support for all steps exists. Conversion to a Markov chain and solution can be automated completely.

Model Simulation vs. Analytical Solutions

- generalized stochastic petri nets are well suited for simulation
- transition rates are not restricted to be deterministic or exponentially distributed
- complex models are computationally expensive (simulation step width and simulation duration)
- too large simulation step width can result in meaningless results (variance of result is too big)

Open Issues of Probabilistic Models

- large gap between system and model
- model construction is time consuming, error prone and unintuitive
- small changes in the architecture result in considerable changes in the model
- model validation for ultra-high dependability

Probabilistic Structural Modeling and Software

- Probabilistic structural based models are not well suited for software. They are rather well suited to analyze hardware architectures and design alternatives.
- for software there are no well defined individual components
- complexity of software structures is very high
- for software the assumption of independent failures is too strong
- one CPU for many processes
- one address range for many functions
- real-time aspects are not captured
- parallelism and synchronization is not considered (except for GSPN’s)

IV.3.3 Reliability Growth Models

- no assumption on identifiable components and system structure
- based on the idea of an iterative improvement process: testing ? correction ? re-testing
- major goals of reliability growth models:
  - disciplined and managed process for reliability improvement
  - extrapolating the current reliability status to future results
  - assessing the magnitude of the test, correction and re-test effort
- allows modeling of wearout and design faults
- can be used for hardware and software as well

Software Reliability Growth Models

- typically continuous time reliability growth
  - the software is tested
  - the times between successive failures are recorded
  - failures are fixed
  - observed execution time data \( t_1, t_2, t_3, \ldots, t_i \) are realizations of the random variables \( T_1, T_2, T_3, \ldots, T_i \)
  - based on these data the unobserved \( T_i, T_i + t, \ldots \) should be predicted (e.g. \( T_i = \text{MTTF} \))

But:

- accuracy of models is very variable
- no single model can be trusted to behave well in all contexts
The Prediction System

Software reliability growth models are prediction systems which are comprised of:

- The probabilistic model which specifies the distribution of any subset Tj's conditional on a unknown parameter a.
- A statistical inference procedure for a involving use of available data (realizations of Tj's)
- A prediction procedure combining the above two points to allow to make probability statements about future Tj's

Musa Model

The basic idea behind the Musa model:

- The software starts with N0 errors, n errors are removed during debugging
- The failure rate is defined by (failure rate = failure correction rate)
- The number of errors after time t is determined by the differential equations
- MTTF = 1/f(t) = (1/iKN0) e^-iKt

Problems with the Musa Model

- does not consider "error size"
- Def.: The size of an error is the probability that an element selected from I results in a failure
- error size usually decreases over time (diminishing returns of heroic debugging)
- assumption of independent inputs is too restrictive (program input is also determined by history)
- assumption of identical failure rates for errors
- assumption that no new errors are introduced (invalid for iterative software development process)
- accuracy is very variable

Error Seeding

- an experimental approach to evaluate the software development processes and testing techniques
- the program P is seeded with m errors (one at a time), and for each error all the test cases are run until the error is detected or the set of test cases is exhausted
- evaluation of the correctness probability:
- evaluation of testing efficiency

Comparison of probabilistic Techniques (1/2)

<table>
<thead>
<tr>
<th>Method simple</th>
<th>Advantages</th>
<th>Restrictions and deficiencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>block diagrams</td>
<td>simple and easy to understand model, way to calculate for constant failure rates</td>
<td>restricted to series and parallel connection, assumption of independent failures, maintenance cannot be modellled, only for non-redundant systems, not well suited for software</td>
</tr>
<tr>
<td>arbitrary block diagrams</td>
<td>can be used to model arbitrary structures</td>
<td>same restrictions as with simple block diagrams, except series and parallel connection, not well suited for software</td>
</tr>
<tr>
<td>markov-chains</td>
<td>can model arbitrary structures, no restriction to independent failures, complex dependencies can be expressed, modeling of coverage and maintenance, good tool support</td>
<td>compared to GPSPH more modular, complex, restriction to constant failure rates, not well suited for software</td>
</tr>
</tbody>
</table>

Comparison of probabilistic Techniques (2/2)

<table>
<thead>
<tr>
<th>Method simple</th>
<th>Advantages</th>
<th>Restrictions and deficiencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>generalized stochastic petrinets</td>
<td>much richer mechanisms for modeling, allows combination of discrete and stochastic behavior, good tool support, can be used to model algorithmic issues of software</td>
<td>computational complexity (seeded errors by number of test cases), error size needs to be controlled</td>
</tr>
<tr>
<td>reliability growth models</td>
<td>suited to prediction of software reliability, does not make assumptions on the system structure, based on relatively easy obtainable experimental data</td>
<td>accuracy of models is very variable, no general applicable model, user must analyze different models to select suitable one</td>
</tr>
<tr>
<td>error seeding</td>
<td>very easy procedure, takes few assumptions on the system</td>
<td>computational complexity (seeded errors by number of test cases), error size needs to be controlled</td>
</tr>
</tbody>
</table>
**Limits of Validation for Ultra-High Dependability**

- $10^6$ catastrophic failure conditions per hour for civil transport airplanes
- experimental system evaluation is impossible for critical applications
- modeling is therefore the only possibility to validate ultra-high dependability

Limits for reliability growth models:
If we want to have an assurance of high dependability, using information obtained from the failure process, then we need to observe the system for a very long time.

Limits of testing:
If we see a period of 10-hour failure-free operation a MTTF of $10^6$ hours can be expected without bringing any a priori belief to the problem. If a MTTF of $10^6$ is required and only 10-hour periods of test are carried out, Bayesian analysis shows that essentially we need to start with a 50:50-belief that the system will attain a MTTF of $10^6$. If we see a period of $10^6$-hour failure-free operation a MTTF of $10^6$-hours can be expected without bringing any a priori belief to the problem. If a MTTF of $10^6$ is required and only 10-hour periods of test are carried out, Bayesian analysis shows that essentially we need to start with a 50:50-belief that the system will attain a MTTF of $10^6$.

This observation is used, but it is impossible to extrapolate from known systems and known methodology.

**Summary**

- modeling is used for the design and validation phase
- deterministic and probabilistic modeling
- probabilistic structural based modeling
- simple block diagrams
- arbitrary block diagrams
- Markov models
- Generalized stochastic Petri nets (GSPN)
- for evaluation of design alternatives
- reliability and safety need to be considered individually
- the interdependence of assumption coverage and system complexity needs special consideration
- single parametric measures (mean time to failure, mean time to repair, mission reliability, availability)
- limits of modeling for ultra-high dependability
- currently there is no methodology available to gain confidence that complex systems guarantee ultra-high dependability
- it is impossible to extrapolate from known systems and known methodology.

**IV.4 Discussion & Summary**

- Specifications and models such as
  - Fault Models (kind of faults)
  - Accident Models (how accidents happen)
  - Formal Models (what does the system?)
  - are used, but
  - effects not covered (abstraction) cannot be analyzed
  - building adequate modeling is crucial

This observation also holds for
- The application of fault tolerance techniques
- Reliability models
IV.5 Bibliography


