Software Fault Tolerance Fundamental Concepts

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University

Overview

(Pullum Chapter 1 / Kienzle 1.4)

- Motivation for Fault Tolerance
- Terminology
  - Faults, Errors and Failures
- Dependability
- Recovery
  - Backward and forward
- Redundancy
- Error Confinement

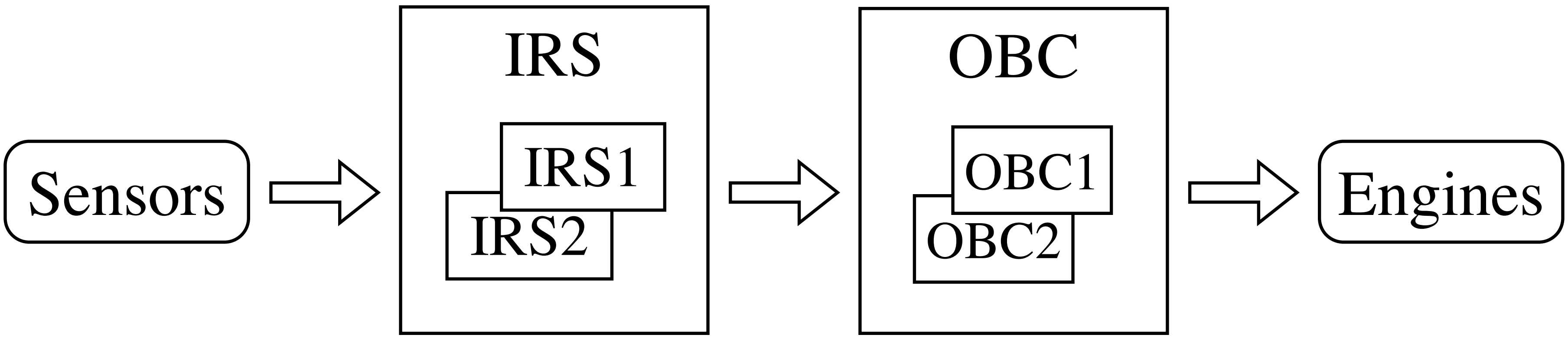
Motivation (1)

- Scope, complexity and pervasiveness of computer-based and controlled systems continue to increase
- Software assumes more and more responsibility
- Consequences of systems failing
  - Annoying to catastrophic
  - Opportunities lost, businesses failed, security breaches, systems destroyed, lives lost

Examples of Software Failures (1)

On June 4, 1996 an Ariane V rocket launched by the European Space Agency exploded just forty seconds after lift-off

Ariane V Architecture



"hot standby"

Ariane V Launch, June 4th 1996

IRS raises an *Operand Error* exception while converting a 64bit float to 16bit integer

No specific exception handler

Operand Error caused by high value of *Horizontal Bias*, which is normal for Ariane V

Function serves no purpose after lift-off in Ariane 5
Ariane IV, from which the code was reused, needs it during 50 seconds

Not possible to switch to backup IRS, for it had failed as well (72ms earlier)

On-board Computer interprets "core dump" data as normal flight data

Full nozzle deflection of solid boosters and vulcan engine

Angle of attack > 20°

Separation of boosters from main stage

Self-destruction after 39 seconds

Examples of Software Failures (2)

- Aerospace
  - Denver airport: Failure in luggage management system
    ⇒ opening delayed for several months
  - Failure of a space probe sent to Mars due to inhomogeneity of measuring units (inch and cm)
  - Launch of Atlantis delayed 3 days
  - Problems when space shuttle Endeavor met with Intelstat 6 due to rounding of near-zero values
  - Flaw in Apollo 11 software made moons gravity repulsive rather than attractive

Examples of Software Failures (3)

- AT&T system suffered a 9 hour US-wide blockade
  - Switch experienced abnormal behavior ⇒ due to flaws in recovery recognition software and network design effects propagated to all switches
- Software problem caused radiation safety door of a nuclear power processing plant in the UK to open accidentally
- Several patients killed through radiation overdoses due to software flaws in Therac-25 (cancer treatment system)

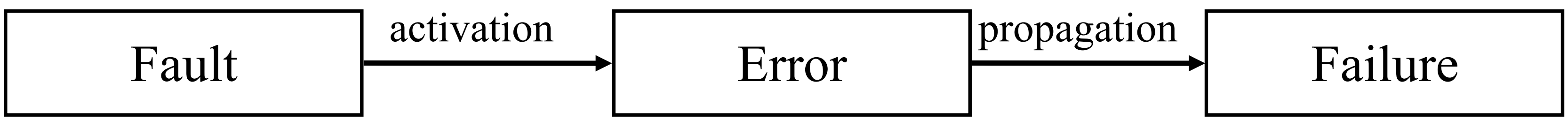
Motivation (2)

- Considerable progress in software engineering
  - Analysis
  - Design
  - Testing
  - Formal methods
  - CASE tools
- Experience shows that we still can not assume that the produced software is fault free

Terminology

- Failure
  - Observable deviation from the specification
- Error
  - Part of the system state that leads to a failure
    - Latent errors [Lap85]
- Fault
  - “Defect” or “Flaw” of a system
  - Bug

Causal Relationship



(Failure ⇒ Error ⇒ Fault)

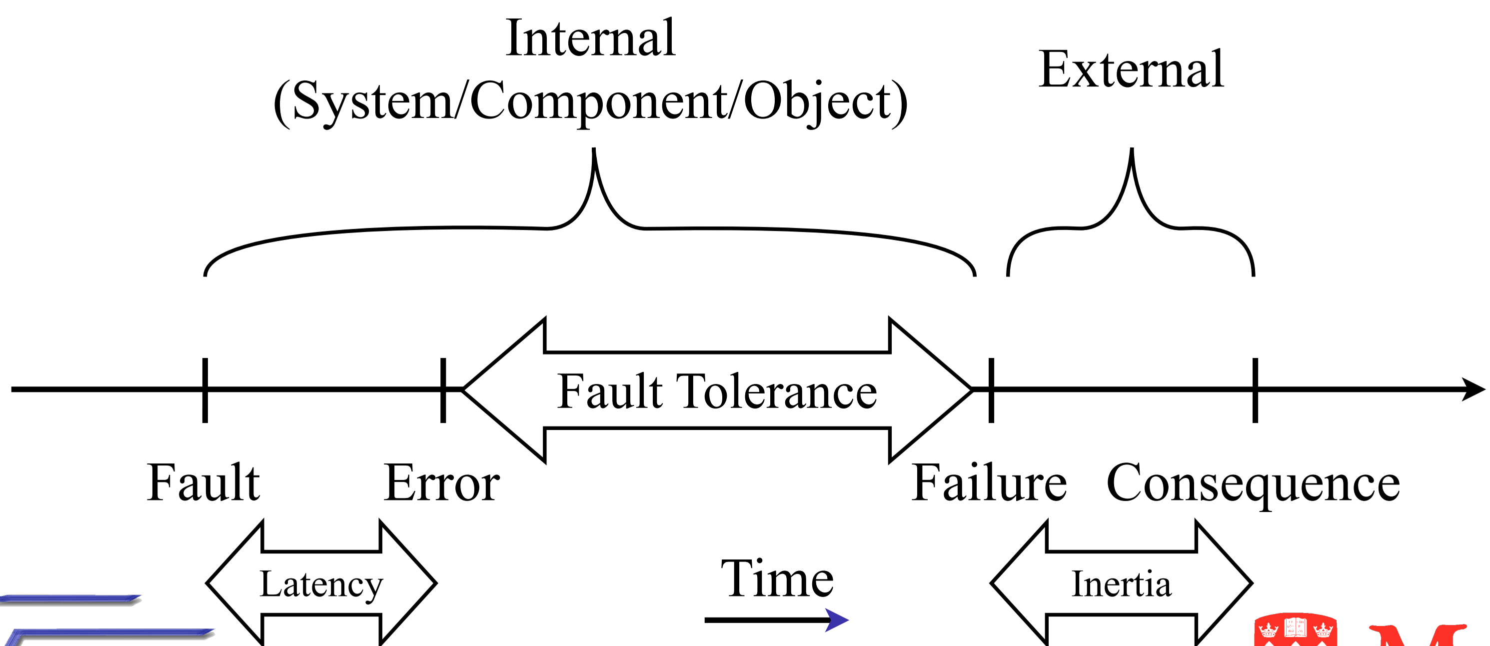
- Hierarchical model
  - Failure at one level can be seen as a fault at a higher level

Goal of Fault Tolerance

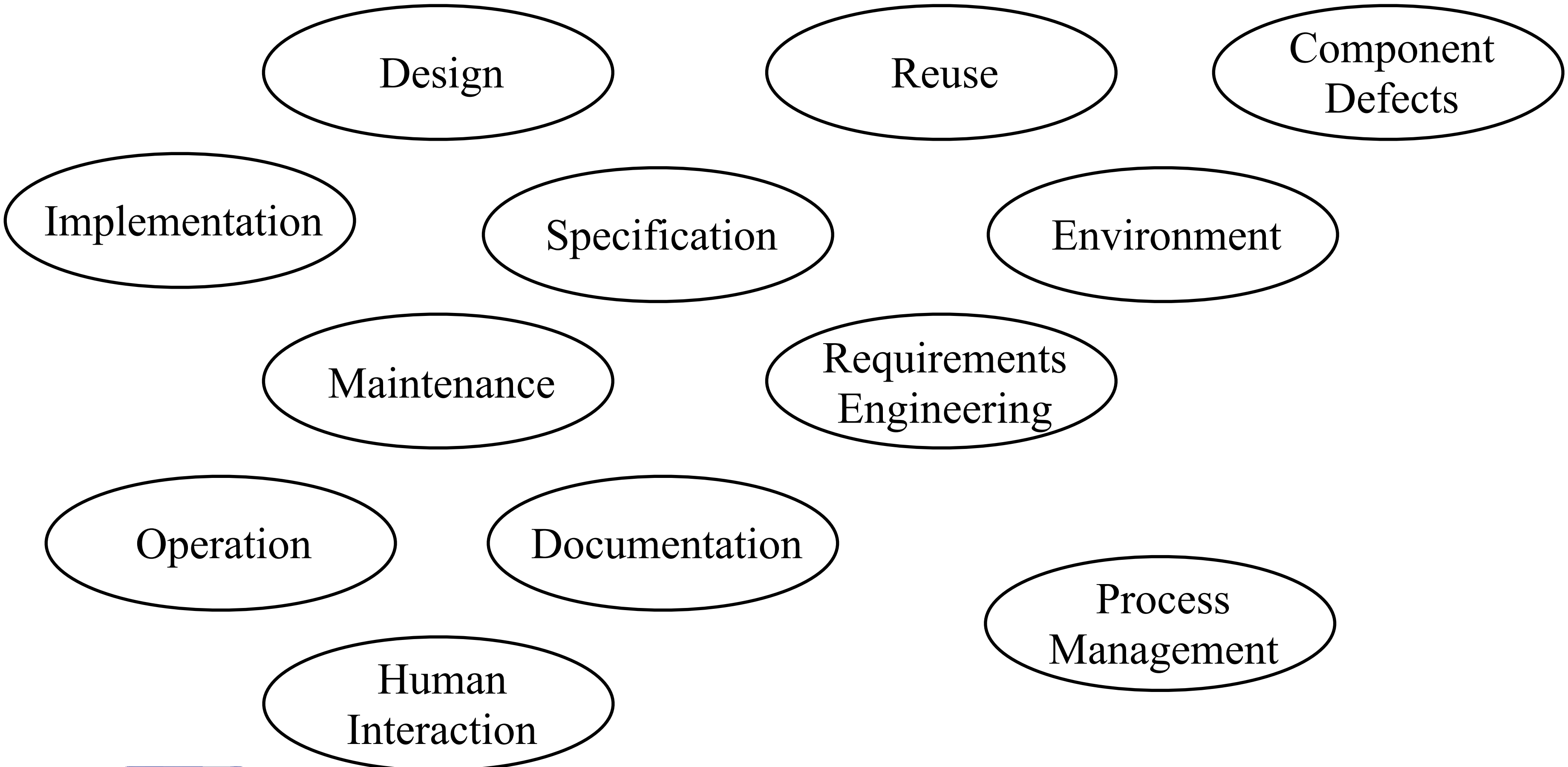
The Goal of Fault Tolerance is to
Avoid System Failure in the Presence of Faults

Fault Tolerance

- Continue to provide service in the presence of faults of underlying components or the environment



Origin of Faults



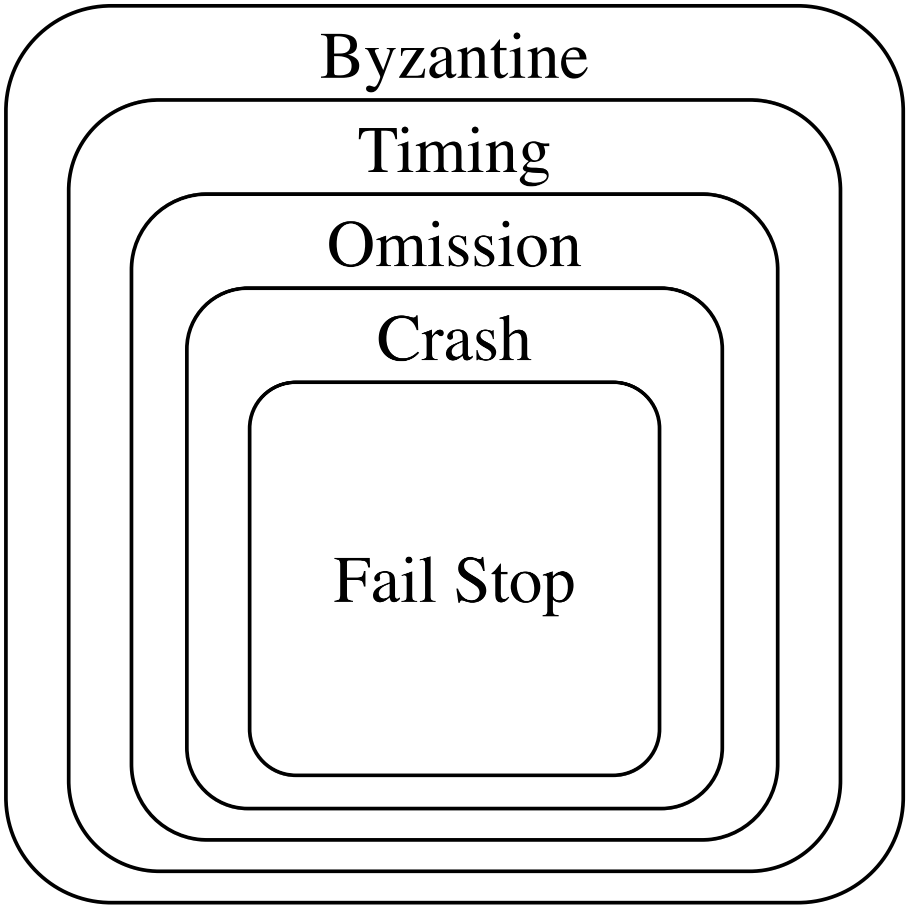
Fault Classification

- Temporal Occurrence
  - Transient fault
  - Intermittent fault (periodic fault)
  - Permanent fault
- Creation time
  - Design fault
  - Operational fault
- Intention
  - Accidental fault
  - Intentional fault

Failure Semantics

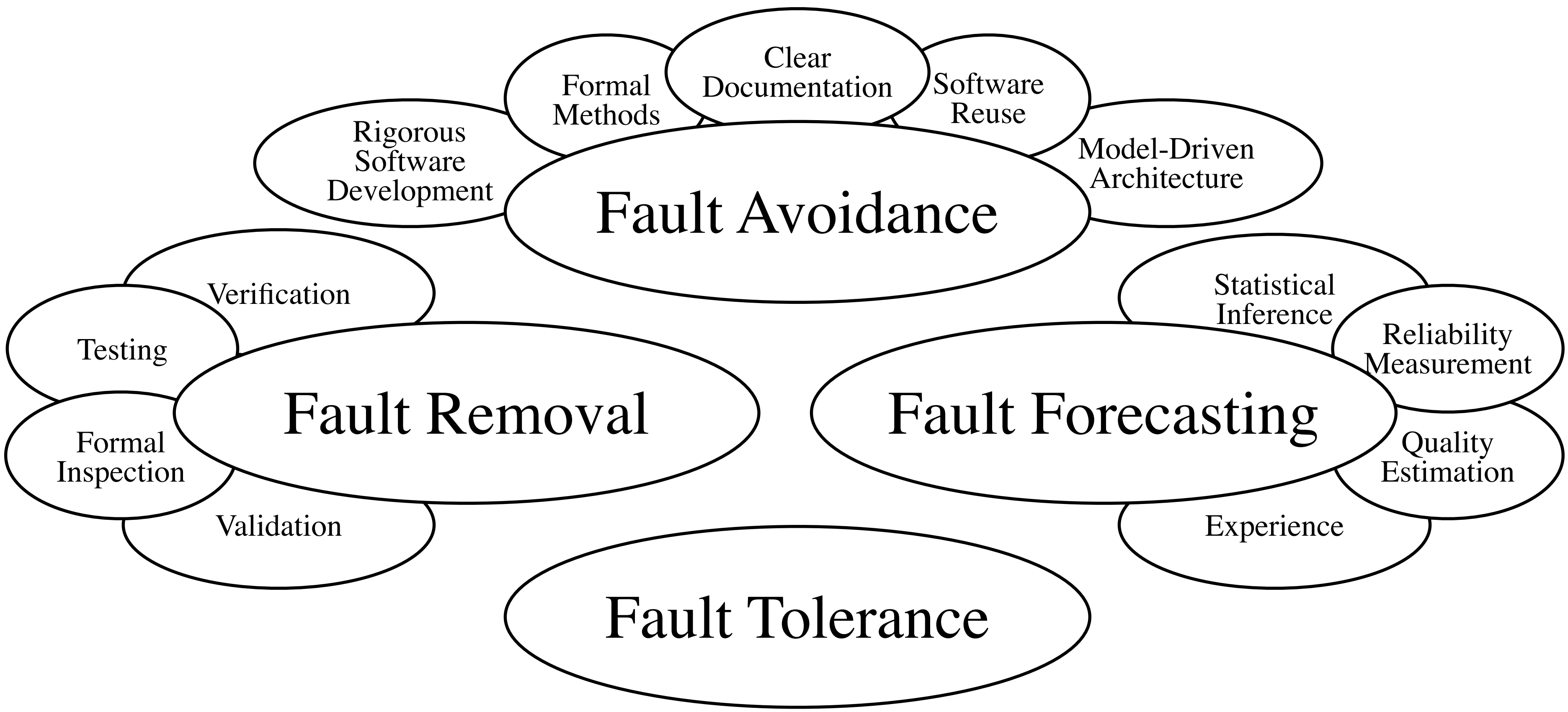
- Crash failure
  - Fail-silent and Fail-stop
- Omission failure
- Timing failure
  - System fails to respond within a specified time slice
  - Both late and early responses might be “bad”
  - Also called performance failure
- Byzantine failure
  - System behaves arbitrarily

Failure Hierarchy



The algorithms used for achieving any kind of fault tolerance depend on the computational model

Reliable Software Development



Fault Avoidance / Prevention

- Reduce the number of faults during software construction
  - Rigorous Software Development Process
    - Requirements Specification & Analysis
    - Structured Design
    - Well-defined mapping to Programming Languages
    - Clear Documentation
  - Formal Methods
  - Software Reuse

Rigorous Software Development (1)

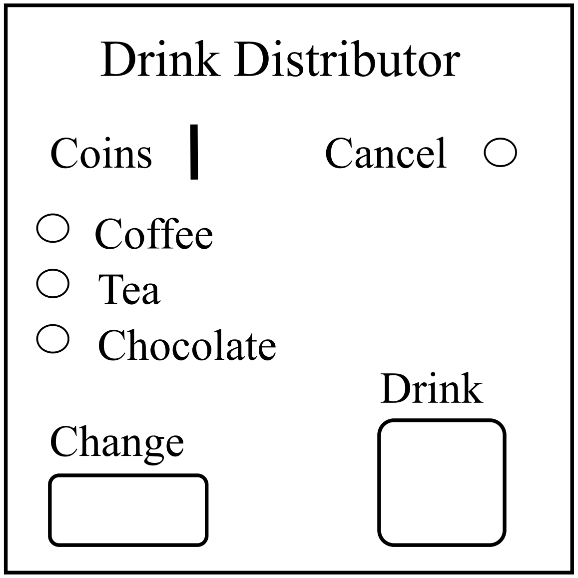
- Requirements elicitation
  - Discover what features each stakeholder expects the system to provide
  - Imperfect process
    - Technical and non-technical people have to collaborate
      - Use-cases
  - Computer scientists can't be experts in all application areas

Rigorous Software Development (2)

- Analysis / Specification
  - Specify in a clear and precise way what functionality your system must provide
  - Complete, but not too complex
  - Consistent
  - Determine (or even better: generate) test cases

Drink Distributor Example (1)

- Provides hot drinks: coffee, tea and chocolate
- User interface
- Cycle treatment
  1. Insert money
  2. Choose drink
  3. Take change
  4. Take drink
  - Or press cancel ⇒ coins are given back



Drink Distributor Example (2)

- Incomplete specification
  - No deadline for cancellation specified
  - What if user inserts new coins before the end of a cycle?
  - What if the user changes his selection?
  - What should be done when resources (change, cups, spoons, sugar, coffee, tea, chocolate, water) run out?
  - Provide partial service?
    (e.g. only tea and coffee / require exact change)
- If manufacturer and user make divergent interpretations, operation time failure will occur

Drink Distributor Example (3)

- Augment specification
  - Cancellation not possible once drink has been chosen
  - Add green / red light to indicate cycle start
  - Only the first selected beverage is taken into account
  - Add lights to show availability of drinks
- Each omission of constraint in the specification can lead to a failure in the service delivered to the user
  - Dissatisfaction
  - Loss of money

Rigorous Software Development (3)

- Structured design
  - For instance in Object-Orientation:
    Apply O-O principles, e.g. abstraction, information hiding, modularity, classification, to reduce complexity of the solution
  - Assign responsibilities to objects
  - Provide easy-to-read documentation
    - UML

Rigorous Software Development (4)

- Programming Methodology
  - Good programming discipline
    - Pair-programming
  - Well-defined mapping of design models to programming constructs
  - Standards or coding conventions

Formal Methods (1)

- Specifications are developed using mathematically tractable languages and tools
  - Petri Nets, Algebraic Specifications
- Allows proving of desired properties
  - Verification and validation
- Generation of test cases
- Generation of code!

Formal Methods (2)

- Mathematical specifications of software tend to be equal in size as the program itself
  $\Rightarrow$ just as error-prone
- Tools (model-checkers) still face algorithmic challenges when attempting to prove properties of huge models
- Have been successfully applied for “small”, safety-critical components
- Domain-specific modeling!

Software Reuse

- Well exercised software is less likely to fail
- Save development cost
- Undiscovered faults may appear when the component is used in a new environment

Fault Removal

- Detect and remove existing faults by verification and validation
- Testing
  - Exhaustive testing not feasible
  - Can’t show the absence of faults
  - Quality measures
- Formal Inspection
- Formal Design Proofs

Fault Forecasting

- Also known as
  Software reliability measurement [Lyu96]
- Estimation
  - Gather failure data during operation or testing
  - Apply statistical inference techniques
- Prediction
  - Gather software metrics during development
- Fault forecasting can indicate the need for additional testing or for applying fault tolerance

Seriousness Classes (1)

- DO-178B, civil aeronautics
  - Without effects
  - Minor / benign
    - Upset passengers, small increase in workload for the crew
  - Major / significant
    - Injuries of the passengers / crew and reducing the efficiency of the crew
  - Dangerous / serious
    - Small number of casualties / serious injuries, or preventing the crew from achieving its task in a precise and complete manner
  - Catastrophic / disastrous
    - Leading to human lives loss

Seriousness Classes (2)

- DO-178B, civil aeronautics
  - Without effects
  - Minor / benign
    - Probable: $p > 10^{-5}$
  - Major / significant
    - Rare: $10^{-7} < p < 10^{-5}$
  - Dangerous / serious
    - Extremely rare: $10^{-9} < p < 10^{-7}$
  - Catastrophic / disastrous
    - Extremely improbable: $p < 10^{-9}$

Software Fault Tolerance

- Tolerate faults that remain in the system after development, preventing system failure
  - ⇒Remove errors and their effects from the computational state before a failure occurs
- Successfully applied in aerospace, nuclear power, healthcare, telecommunications and transportation industries
- 35 years of research

Classification

- Single Version Software
  - Monitoring techniques, atomicity of actions, decision verification, exception handling
- Multi-version Software
  - Functionally independent, yet equivalent software
  - Recovery blocks, N-version programming, …
- Multiple Data Representation
  - Retry blocks, N-copy programming, …

Recovery

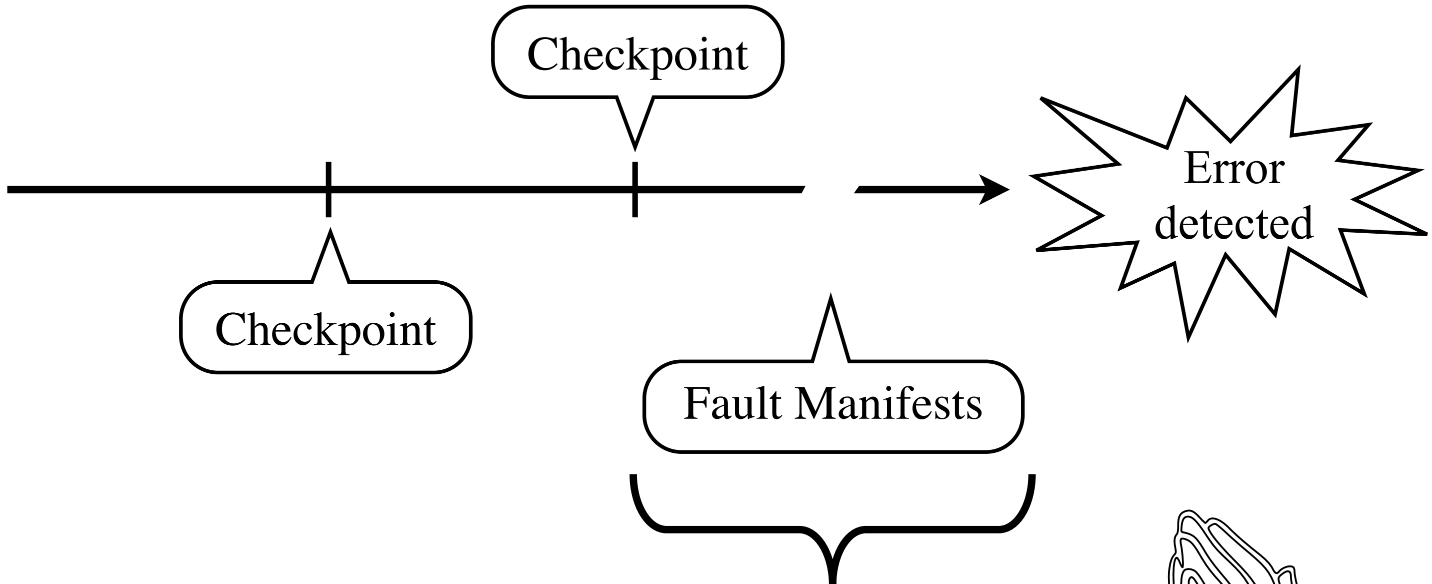
- Error detection
  - Identify erroneous state
- Error diagnosis
  - Assess the damage
- Error containment / isolation
  - Prevent further damage / error propagation
- Error recovery
  - Substitute the erroneous state with an error-free one
- Backward and Forward Error Recovery

McGill

Backward Error Recovery (1)

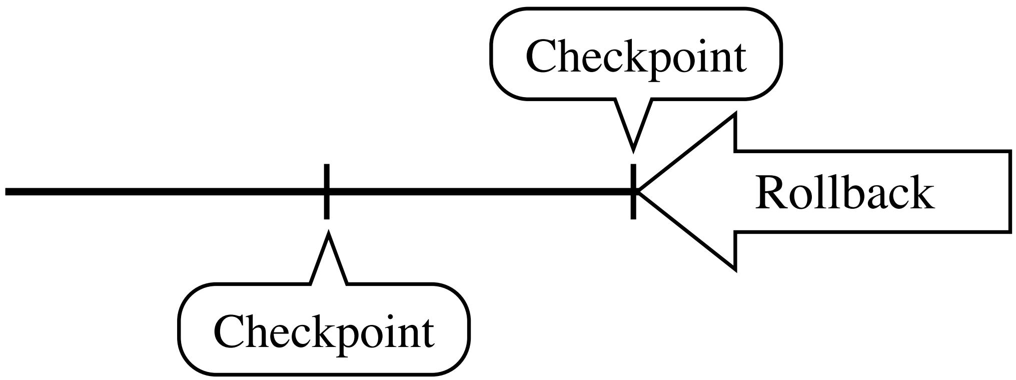
- System state is saved at predetermined recovery points
  - Called checkpointing
  - Incremental checkpointing, log
- State should be checkpointed on stable storage, not affected by failures
- Recover error-free state by rolling back to a previously saved (error-free) state

Backward Error Recovery (2)



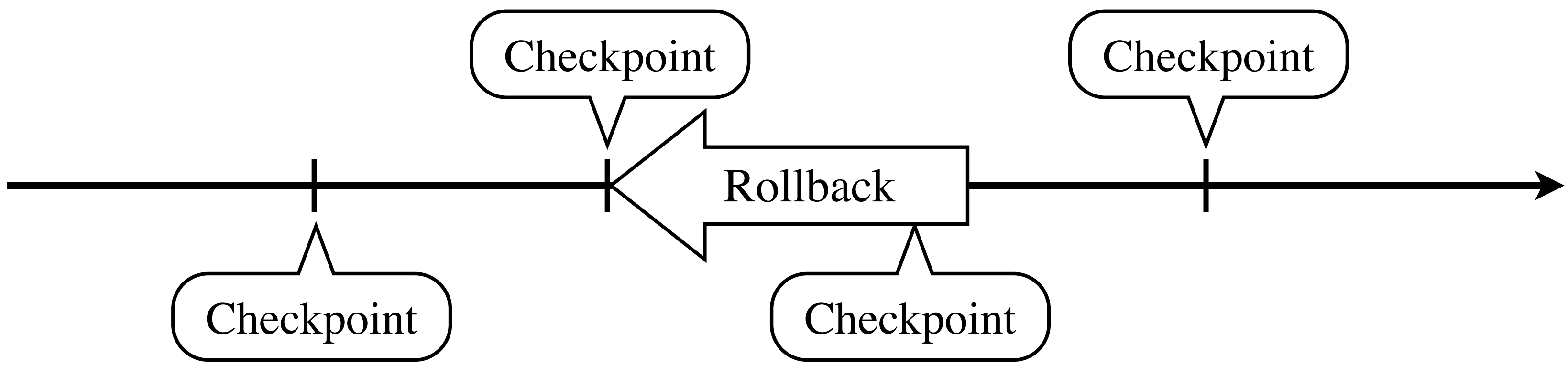
Assumption:
Faulty behavior occurred
after last checkpoint

Backward Error Recovery (2)



Assumption:
Faulty behavior occurred
after last checkpoint

Backward Error Recovery (2)



Depending on the assumed fault and on the specific fault tolerance technique used:

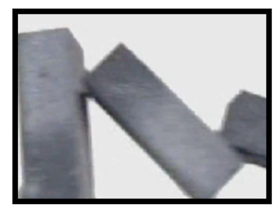
- Try again
- Try a different alternate
- Do nothing (wait for the next request)

Advantages of Backward Recovery

- Requires no knowledge of the errors in the system state
- Can handle arbitrary / unpredictable faults (as long as they do not affect the recovery mechanism)
- Can be applied regardless of the sustained damage (the saved state must be error-free, though)
- General scheme / application independent
- Particularly suitable for recovering from transient faults

Disadvantages of Backward Recovery

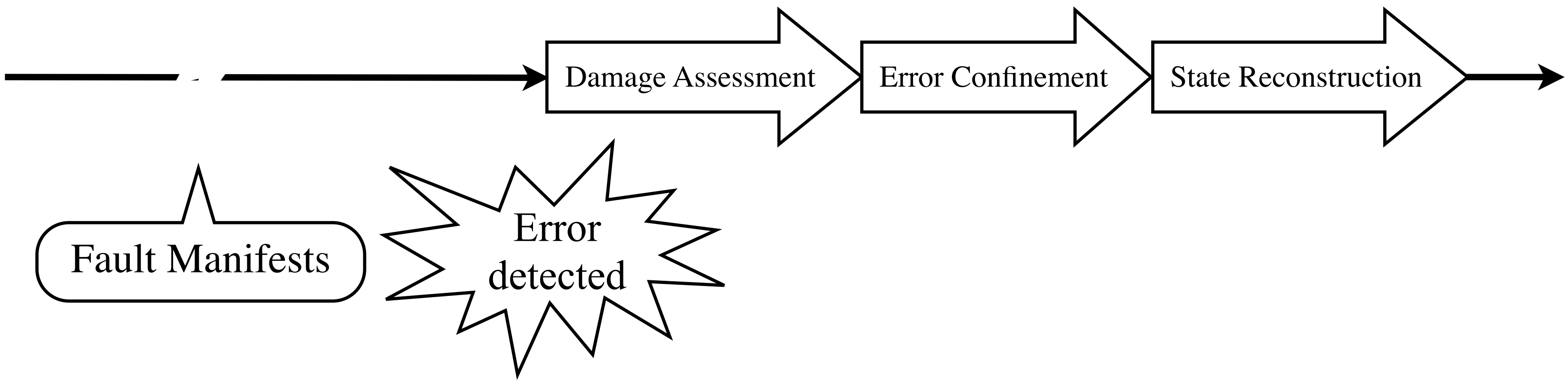
- Requires significant resources (e.g. time, computation, stable storage) for checkpointing and recovery
- Checkpointing requires
  - To identify consistent states
  - The system to be halted / slowed down temporarily
- Care must be taken in concurrent systems to avoid the domino effect



Forward Error Recovery

- Detect the error
- Detailed damage assessment
- Build a new error-free state from which the system can continue execution
  - "Safe stop"
  - Degraded mode
  - Error compensation
    - E.g., switching to a different component, etc…

Forward Error Recovery (2)



Advantages of Forward Recovery

- Efficient (time / memory)
  - If the characteristics of the fault are well understood, forward recovery is the most efficient solution
  - Well suited for real-time applications
  - Missed deadlines can be addressed
- Anticipated faults can be dealt with in a timely way using redundancy

Disadvantages of Forward Recovery

- Application-specific
- Can only remove predictable errors from the system state
- Requires knowledge of the actual error
- Depends on the accuracy of error detection, potential damage prediction, and actual damage assessment
- Not usable if the system state is damaged beyond recoverability

Redundancy

- Key concept of fault tolerance
  - Hardware redundancy
    - Most common use of redundancy
    - We're not going to address it
  - Software redundancy
    - Additional applications, modules, objects used in the system to support fault tolerance
  - Information redundancy
    - Error-detecting or error-correcting codes
    - Diverse data
    - Data produced for fault tolerance
  - Time redundancy
    - Use additional time for fault tolerance

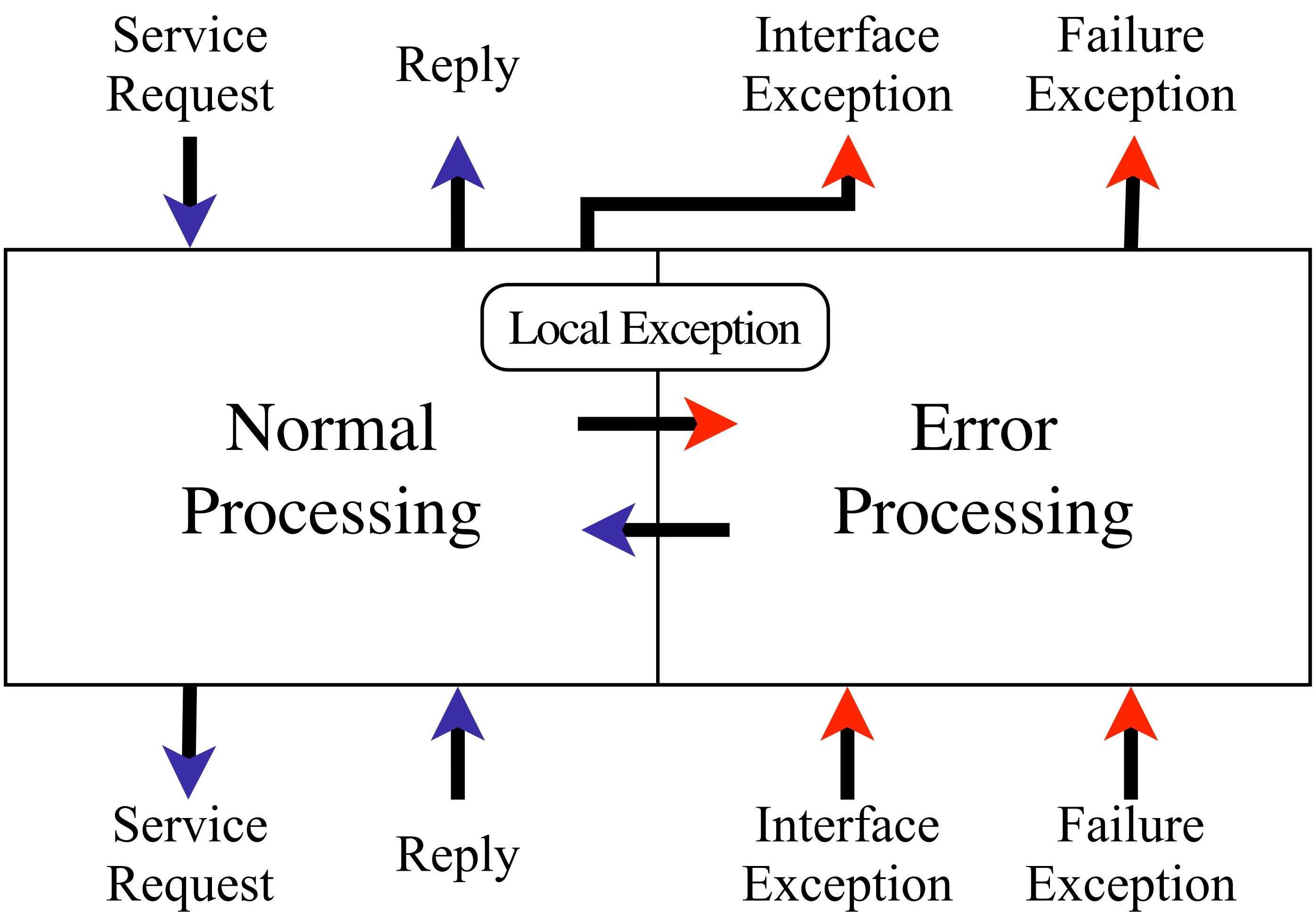
Architectural Structure

- Systems, especially concurrent ones, are increasingly complex
- Consist of several components / subcomponents
- Fault tolerance must account for that
  - Different fault tolerance approaches for each components
  - Failure of a subcomponent can be perceived as a fault in the parent component
- Clear structuring reduces complexity

Error Confinement

- System partitioned into regions, beyond which effects of faults should not propagate
- Components should only be accessible through a well-defined (and preferably narrow [Kop97]) interface
- Different confinement regions may employ different fault tolerance techniques depending on failure semantics of the environment and subcomponents

Idealized Fault-Tolerant Component [Lee90]



Idealized Fault-Tolerant Component

- Receives requests for service
- Produces responses
- 3 kinds of exceptions
  - Interface exception: An invalid service request has been made
  - Local exception: An internal error is detected
  - Failure exception: Component is unable to provide the requested service
- Recursive structure

Questions

- What are the four means for achieving dependability?
- What is the goal of software fault tolerance?
- Name the two error recovery strategies, and briefly explain how they work…
- What are the different forms of redundancy that can help constructing fault tolerant software?
- What are latency and inertia?

References

- [Lap85]
  Laprie, J.-C.: “Dependable Computing and Fault Tolerance : Concepts and Terminology”, in *Proceedings of the 15th International Symposium on Fault–Tolerant Computing Systems (FTCS–15)*, pp. 2 – 11, Ann Arbour, MI, USA, June 1985
- [Lyu96]
  Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, New York, IEEE Computer Society Press, McGraw-Hill, 1996.
- [Kop97]
  Kopetz, H.: *Real–Time Systems — Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
- [RX95]
  Randell, B.; Xu, J.: The Evolution of the Recovery Block Concept, chapter 1, pp. 1 – 21, in Lyu, M. R. (Ed.): *Software Fault Tolerance*, John Wiley & Sons, 1995.

McGill