Dependable and Distributed Systems

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 2 - Dependability Concepts

Dependability

A system is said to be **dependable** if it is trustworthy enough for reliance to be justifiably placed on the services it delivers

How trustworthy should such a system be?

How do we measure trustworthiness?

How can we ensure trustworthiness by design?

Trustworthiness

How can we define trustworthiness for dependability?

If your computer crashes once per year, is it trustworthy? Once per day? Once per hour?

Really depends on the system requirements

Specification defined **what the system is supposed to do**

Can we relate trustworthiness and system specification?

If a system fulfils its specification, is it trustworthy?

What Do We Need To Develop Dependable Systems

We have a **system model**

How we view our system - black box, white box, interconnected, etc.

Define a **fault model**

What are the potential problems that we must consider

System has a **specification**

Essentially an oracle for system behaviour

System Model

A system is a set of **elements** that work together to provide service to a **user**

> Elements can be hardware, software, human, etc., whilst user can mean another system, human, etc.
>
> Interaction through well-defined interfaces, e..g, GUI connects human and software, remote procedure call connects software elements

An element is an entity that provide a **predefined service** and is able to communicate with other elements

Depending on the analysis requirements, we can view an element at varying levels of abstraction, e.g., memory or individual registers in hardware, method or object in software

How Do Elements Work Together?

They interact in a well-defined way

Via well defined interfaces

Examples include:

- GUIs (human and software)
- Remote procedure calls (software and software)
- Distributed systems - send, receive, broadcast vs. shared memory (software and software)

What is a Service?

A service is the behaviour that a user perceiver as there system interface

For trustworthy systems, services need to be **correct** and **timely**

What Are The Problems?

A system is a collection is elements

Elements may start failing

- Services not provided as expected, e.g., due to bugs in software
- Hardware problems, e.g., due to ageing mechanical components
- Unexpected circumstances, e.g., spikes in workload

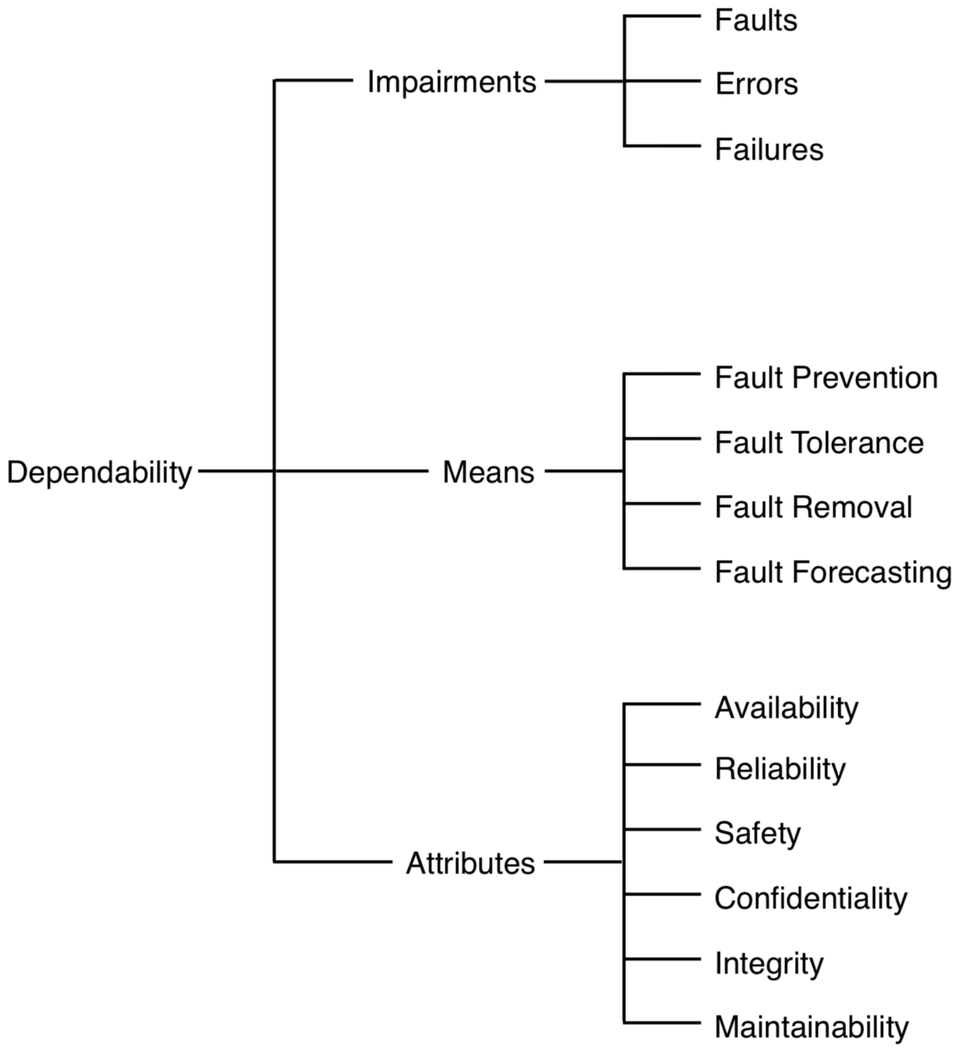
The Dependability Tree

A taxonomy generalises and structures our consideration of dependability

Impairments - What endangers the dependability of a system

Means - What we can we do to impart dependability

Attributes - What measures we uses to establish the dependability of a system



Dependability Impairments

Failure

A system failure occurs when the services provided by a system deviate from its specification

It is observable, for example:

Specification: User must receive an update sensor value every second

Problem: At some point in time, more than one second elapses between successive sensor readings

Conclusion: A failure occurred, since this is an observable deviation from specification

Error

An error is an erroneous system state that can lead to a failure

May be detectable

Not all errors lead to a failure

To prevent failure, we would have to be sure no errors exist

Fault

An error is the consequence of a fault occurring

- It is said to be an undesirable event
- Hypothesised cause of an error
- If no fault occurs, no error can exist

Many possible type of fault

- Software design flaws (bugs), noisy message losses, bit flips, etc.

Fault

Hardware faults are classified with respect to duration as permanent, transient or intermittent

Permanent - Remains active until a corrective action is taken

Transient - remains active for a short period of time

Intermittent - A transient fault that becomes active periodically

Their short duration mean transient faults are usually detected as errors that result from propagation

Transient faults are the dominant type in integrated circuits, e.g., ~98 % of RAM) faults are transient

Fault-Error-Failure Cycle

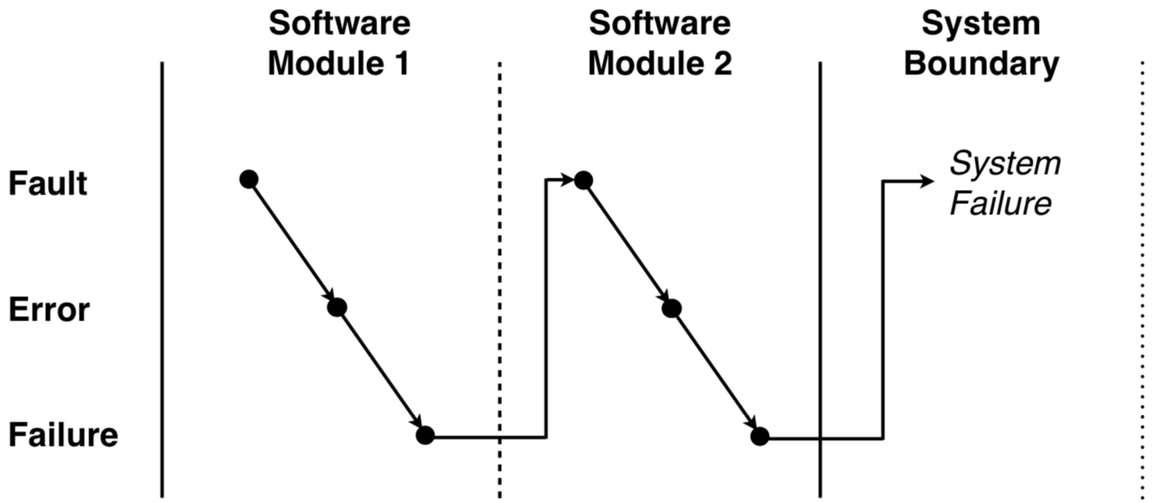
Also known as The Fundamental Chain

Dependability attributes are impacted by the interplay of impairments, i.e, faults, errors and failures, at various levels of abstraction

Recursive structure

Failure at one level of abstraction can represent a fault at another

Leads to the definition extended chains of causality to represent the error propagation



$$fault \rightarrow error \rightarrow failure$$

$$\ldots fault \rightarrow error \rightarrow failure \rightarrow [fault \rightarrow error \rightarrow failure] \rightarrow fault \ldots$$

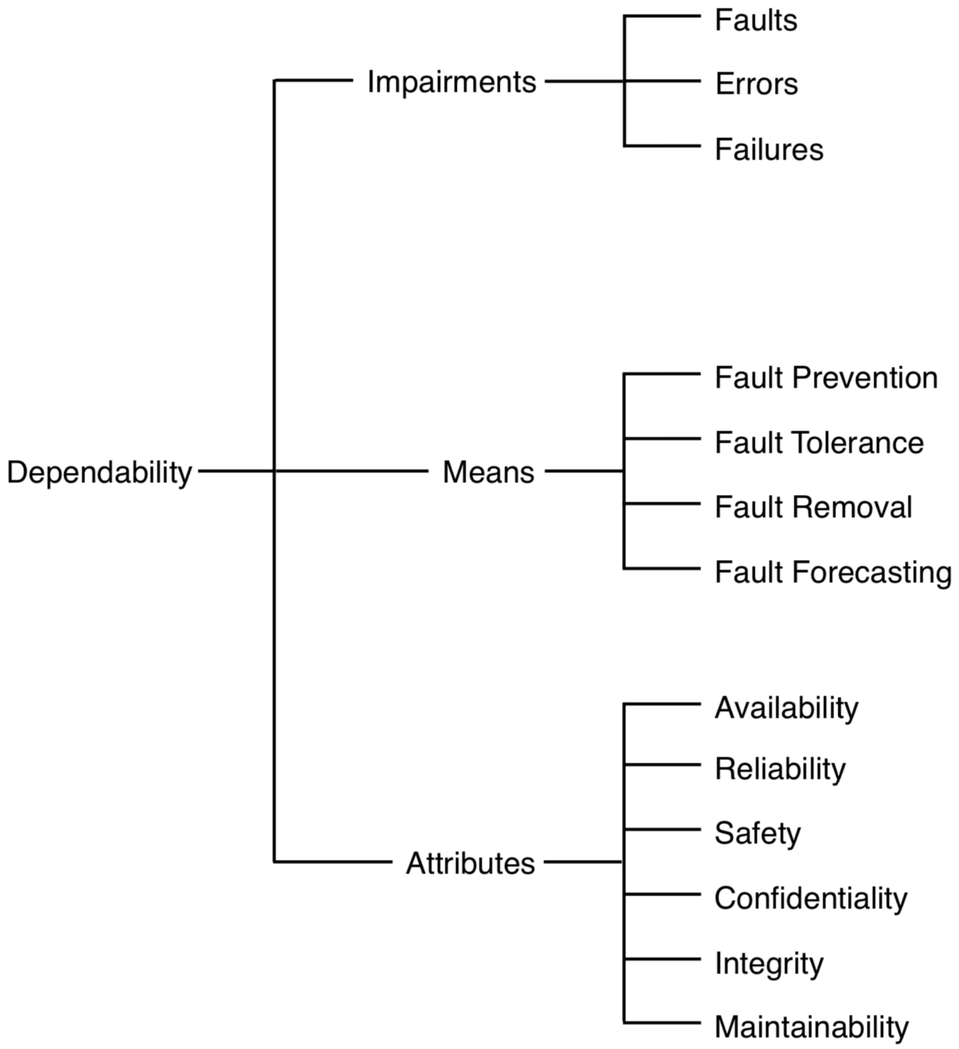
The Dependability Tree

A taxonomy generalises and structures our consideration of dependability

Impairments - What endangers the dependability of a system

Means - What we can we do to impart dependability

Attributes - What measures we uses to establish the dependability of a system



Dependability Means

Dependability Means

Four dependability means that may be employed in combination to provide dependability attributes in the presence of dependability impairments

Fault Prevention

Fault Tolerance

Fault Removal

Fault Forecasting

There is no silver bullet, hence it is common to combine approaches in pursuit of dependability

Fault Prevention

Intention of fault prevention is to hinder and obstruct the occurrence, introduction and spread of faults

Focus on eliminating the conditions for faults to be activated

Examples of fault prevention techniques include modular software design, software development methodologies and process quality assurance

Fault Removal

Intention of fault removal techniques is to reduce the number, likelihood of activation and wider consequences of faults in a computer system

Typically fault removal is a three stage process

Validation - Determine whether a system adheres to a set of defined properties

Diagnosis - Identify problems, i.e., faults, preventing these properties from being fulfilled

Correction - Modify the system to allow the defined properties to be fulfilled

Examples of fault removal techniques include debugging and code reviews

Fault Forecasting

Concerned with estimating the number, likelihood of activation and wider consequences of faults in a computer system

Fault forecasting typically involves the identification, classification and analysis of modes by which a computer system can fail, as well as an evaluation of dependability attributes using probabilistic and analytical approaches

Fault injection analysis is a commonly adopted approach when attempting to establish dependability measures and forecast fault proneness

> Dependability validation approach whereby the response of a system to the artificial insertion of faults or errors is analysed so that insights can be gained regarding the dependability of the system

Fault Tolerance

Concerned with actively handling the occurrence of faults and errors to ensure that a system is able to meet its specification regardless of dependability impairments

Techniques generally focus on the recognition of an erroneous state in a system and restoring a suitably correct state, or at least a safe state, following the occurrence of an error

Widely used means for imparting dependability because we can never guarantee a fault-free system

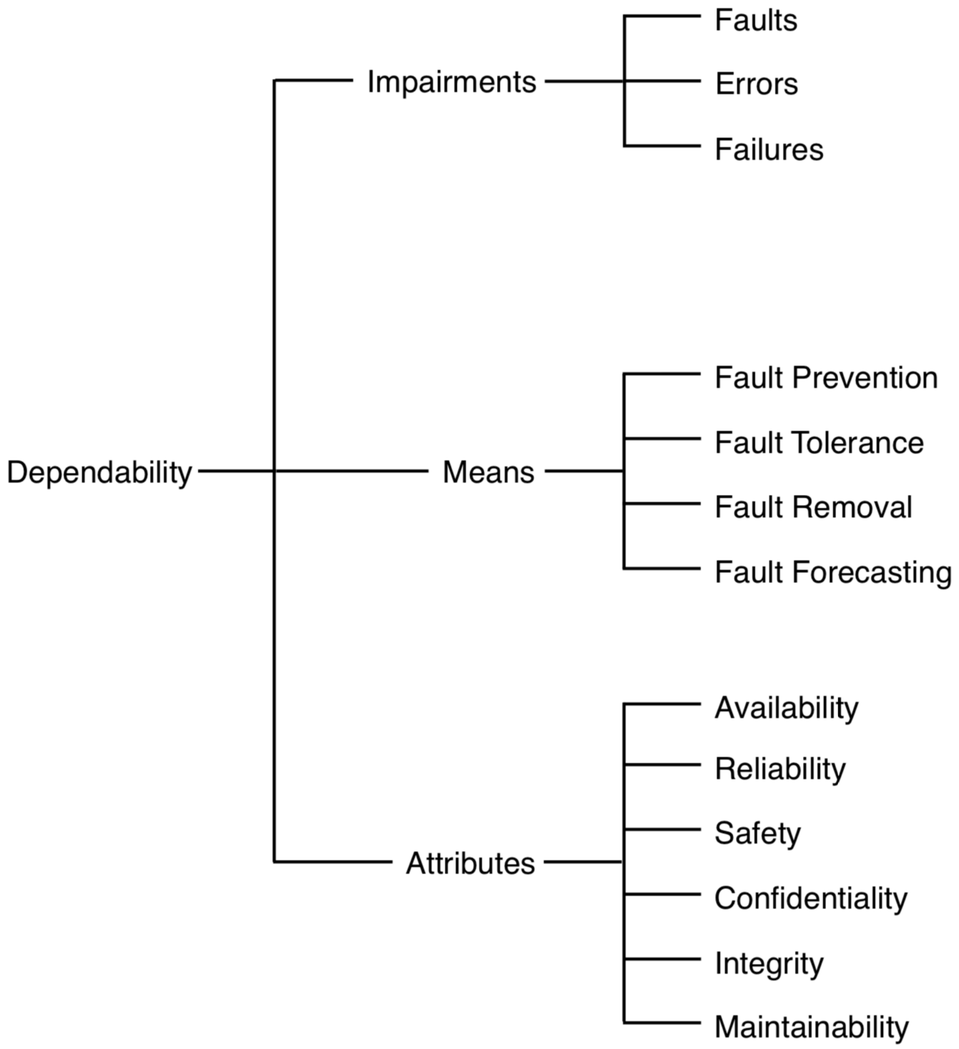
The Dependability Tree

A taxonomy generalises and structures our consideration of dependability

Impairments - What endangers the dependability of a system

Means - What we can we do to impart dependability

Attributes - What measures we uses to establish the dependability of a system



Dependability Attributes

Dependability Attributes

Different application domains commonly require a focus of different attributes of dependability

- Safety-critical domains - high reliability
- Internet services - high availability

Pragmatically, we focus on a subset of attributes for every system we develop

Reliability

Probability of a system to correct services over a specified time period

> For example, an aircraft need high reliability during flights but you don't care quite so much when it is inactive on the ground

More formally, defined as the conditional probability that the system will provide correct service throughout the interval $\mathbf{[t_0,t]}$, given that the system was providing correct service at time $\mathbf{t_0}$

It is typically assumed that time $\mathbf{t_0}$ is the current time, hence **R(t)** is conventionally used to denote reliability

The unreliability of a computer system is conventionally denoted by $\mathbf{Q(t) = 1 - R(t)}$

Reliability - Regimes

Fail-safe

Fail-operational

Fail-secure

Fail-silent

Fault-tolerant

Fail-safe Systems

Fail-safe systems become safe when they can not operate

For example…

Infusion pumps can fail but, as long as the pump complains to the nurse and ceases pumping, it will not be a threat to life because its safety interval is long enough to permit a human response

Industrial and domestic burner controllers (like the ones in our home and workplaces) would be capable of exploding and poisoning humans, if these were not designed to turn off combustion when faults occur

Fail-operational Systems

Fail-operational systems continue to operate when they fail

Examples include train signal, elevators, gas thermostats, passively safe nuclear reactors, etc.

Fail-operational mode is sometimes unsafe

> Nuclear weapons launch-on-loss-of-communication was rejected as a control protocol by the US Military because it is fail-operational - too much risk to human life

Fail-secure Systems

Fail-secure systems maintain maximum security when they can not operate

Examples include fail-secure electronic doors, which revert to being locked when power is removed - potentially devastating consequences depending on context

Fail-silent Systems

Fail silent systems (commonly individual nodes) either function correctly or stop functioning after an internal failure is detected

Ease the design of fault tolerant systems

Can be implemented using commercial-off-the-shelf components and well well-established protocols

Fault-tolerant Systems

Fault-tolerant system continue to operate correctly when subsystems operate incorrectly

For example, autopilot systems on commercial aircraft and control system for ordinary nuclear reactors

A typical method to tolerate faults is to have several computers continually test the part of a system, and switch in a hot spare for failing subsystems

Applicability depends on the fault model assumed

Availability

Defined as a function of time representing the probability a service provided by a system is operating correctly and able to perform its designated function at a given time

Intuitively, the higher the availability of a service provided by a computer system, the more likely that it is to be available when requested

Generally applicable to business-critical systems

Financial services, Internet services, telecommunications

Availability (%)	Downtime
90	36.5 days/year
99	3.65 days/year
99.9	8.76 h/year
99.99	52 min/year
99.999	5 min/year
99.9999	31 s/year

Availability

Availability $A(t)$ of a system at time t is the probability that the system is functioning correctly at time t

$A(t)$ is also referred as **point availability** or **instantaneous availability**, such that it is necessary to determine the interval or mission availability, as defined by:

$$\frac{1}{T}\int_{T}^{0} A(t)dt$$

It is often the case that, after some initial transient effect, the point availability assumes a time-independent value, allowing the **steady-state availability** to be defined by:

$$\lim_{T\to\infty} \frac{1}{T}\int_{T}^{0} A(t)dt$$

Availability

Availability of a service provided by a computer system can be approximated by the ratio of the total time that the computer system has been capable of providing its designated services correctly to the total time that the system has been operational

This means that the steady-state availability of a service provided by a system can also be specified as:

$$A_{steady} = \frac{MTTF}{(MTTF + MTTR)}$$

This is where **MTTF** and **MTTR** terms represent the mean time to failure and the mean time to repair for the service respectively

Steady-state availability is often specified in terms of downtime per year

Safety

The safety attribute of dependability reflects the extent to which a system can operate without damaging or endangering its environment

Some systems have “safe” statuses

- Railways systems where all lights are red, such that all trains halt
- Fly-by-wire avionics is not inherently fail-safe, though it can be transformed to be fail-safe under a reasonable set of failure mode assumptions

Confidentiality

The confidentiality attribute is concerned with the non- disclosure of undue information to unauthorised entities

Serves as a measure of the extent to which a computer system will allow those without sufficient privilege to obtain information that should be not be made available

Integrity

The capacity of a computer system to ensure the absence of improper system alterations, with regard to the with- holding, modification and deletion of information

Typically interpreted such that “improper system alterations” relates only to information alterations performed by an unauthorised entity, though it also en- compasses acts where an unauthorised party prevents modifications or causes information corruption

Maintainability

Defined as a function of time representing the probability that a failed computer system will be repaired in ***t*** time or less

Maintainability attribute is conventionally denoted by **M(t)**

Where a constant rate of repair, μ, can be assumed, the maintainability of a given system can be estimated as:

$$M(t) = 1 - \exp^{-\mu t}$$

What About Security?

Is security a part of dependability?

The composite of the availability, confidentiality and integrity attributes was traditionally considered to account for the computer system security aspect of computer system dependability

Some argue that security considerations have outgrown this interpretation

Software Fault Tolerance

Motivation

Software increasingly defining the functionality of most systems

- Embedded systems, web systems, etc.
- More flexible, maintenance, etc.

Problems arise

- Very difficult to develop demonstrably bug-free (fault-free) software
- Faults at lower levels, e.g., hardware and middleware, can impact on application-level software functionality and performance.
- We use dependability means to tackle problems at software level

Dependability Means Revisited

We have seen the four dependability means in our taxonomy

Fault Prevention, e.g., formal methods

Fault Tolerance, e.g., testing

Fault Removal, e.g., fault injection

Fault Forecasting, e.g., run-time checks

Even though we spend time to perform formal verification, the real program may still be buggy, or verification performed under wrong fault assumptions

We must be able to live with problem, i.e., we need to accept that bad events will happen

Software Fault Tolerance

Given a specification $SPEC$, and a program P that satisfies $SPEC$, and a fault model F

Consider the fault model F to be a function that transforms P into a program $P' = F(P)$

P' is said to be F-tolerant if P' satisfies $SPEC$

Now, we know from our formal methods that a system specification consists of two parts:

Safety

Liveness

Software Fault Tolerance

What if P' satisfies only safety or liveness but not both?

If P' satisfies *safety only* in presence of F, then P' is said to be **fail-safe F-tolerant**.

If P' satisfies *liveness only* in presence of F, then P' is said to be **non-masking F-tolerant**.

If P' does *not satisfy either safety or liveness* in presence of F, it is called **F-intolerant**.

F-tolerant is also known as **masking F-tolerant**

Tolerance Hierarchy

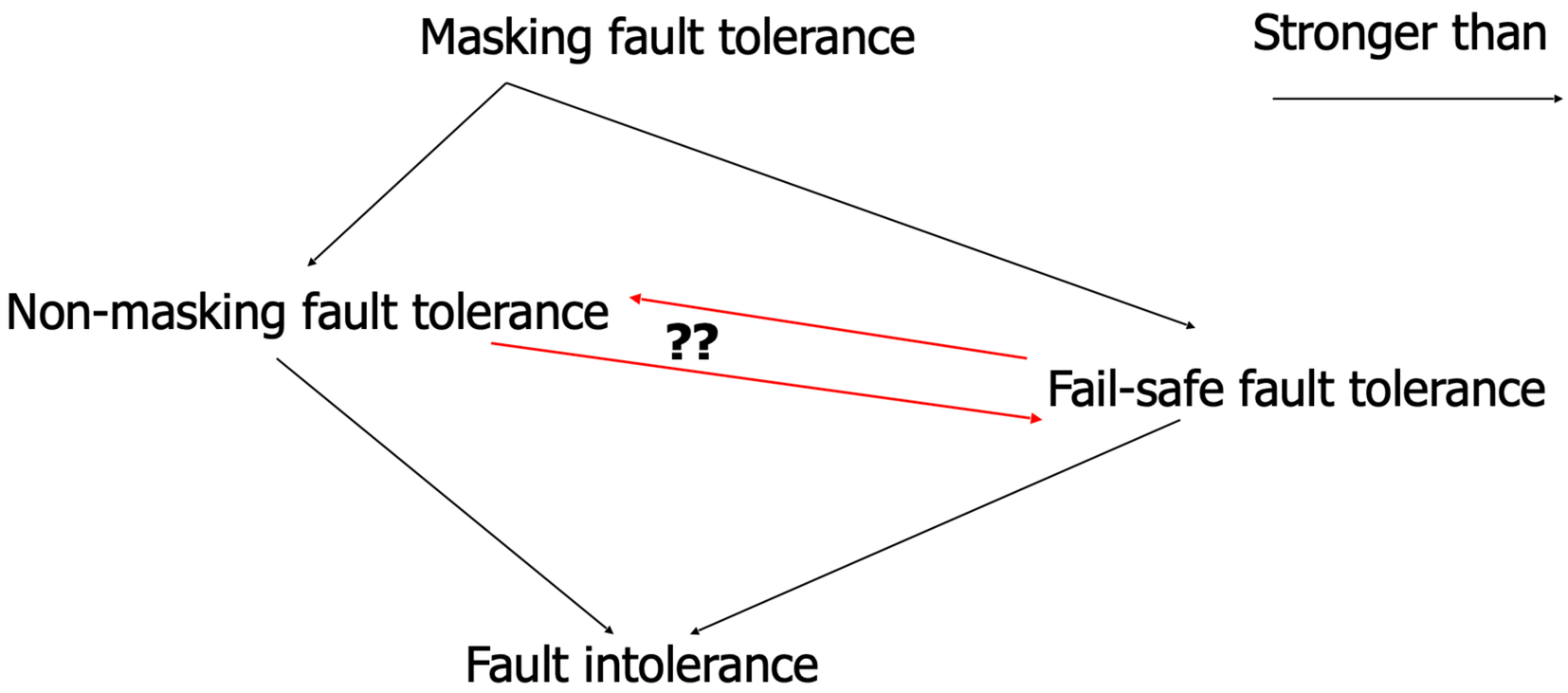
We can see that some of the fault tolerance classes can be ranked

Masking fault tolerance is strictly stronger than non- masking fault tolerance (since the former satisfies both safety and liveness)

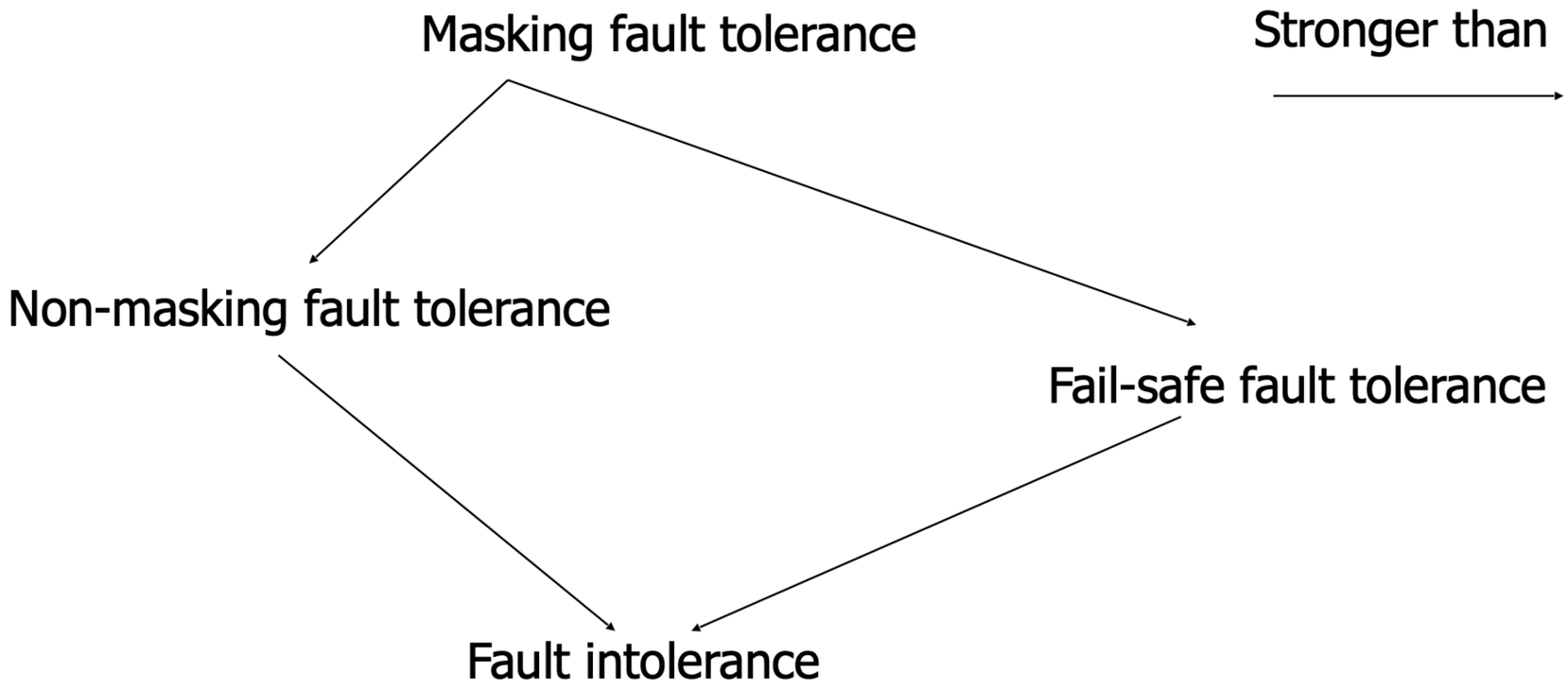
Masking fault tolerance is strictly stronger than fail-safe fault tolerance

How does non-masking and fail-safe rank against each other?

Tolerance Hierarchy Pictorially



The Partial Order Among Fault Tolerance Classes



Notes On Fault Tolerance Classes

What we have talked about up until now can be applied to hardware too

How can we enforce fail-safety or non-masking fault tolerance?

Is it possible to design program such that these properties are enforced?

If yes, how? What do we need?

Program Components

To satisfy safety and liveness, we need only two program components

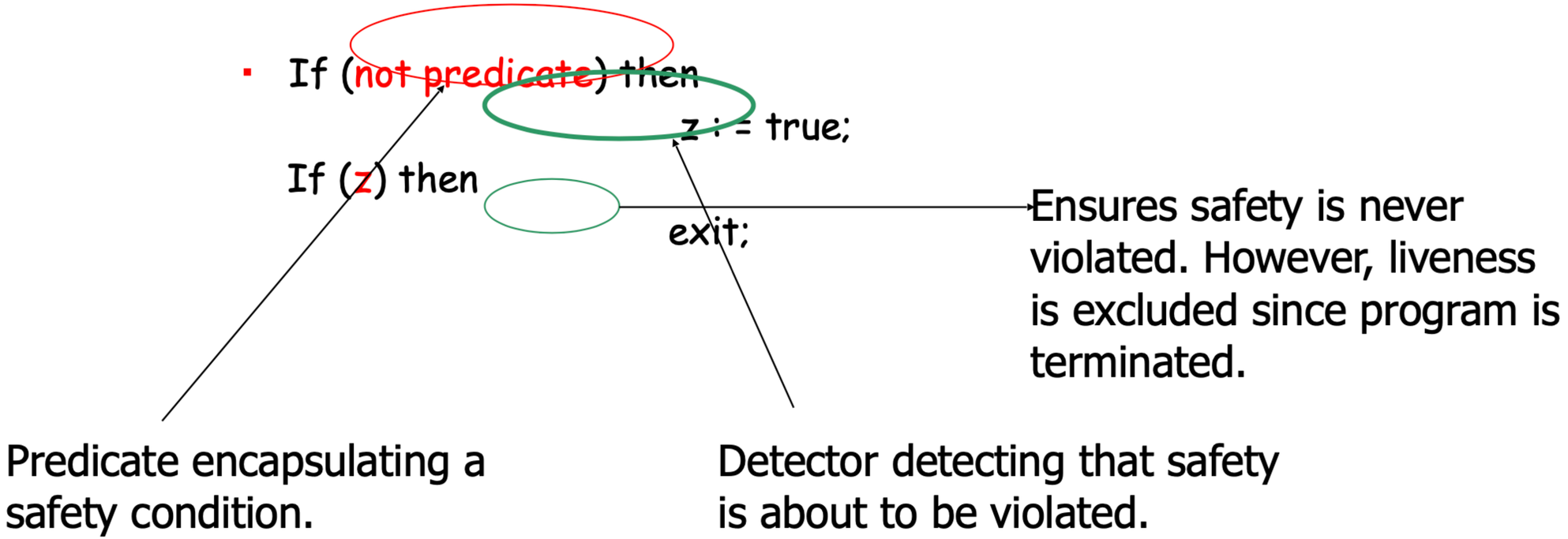
Detectors

Correctors

A **detector** is a class of program components that asserts the validity of a predicate in a running program.

A **corrector** is a class of program components that imposes a given predicate on a running program

Detectors


If (not predicate) then
z := true;
If (z) then
exit;
Ensures safety is never violated. However, liveness is excluded since program is terminated.
Predicate encapsulating a safety condition.
Detector detecting that safety is about to be violated.

Correctors

In its simplest form:

- If (not safety) then
  enforce predicate;

Predicate enforcement techniques ensure that invariant is satisfied again, after it has been violated. Remember invariant captures the safety notion for the program.

Hence, correctors try to reinstate the invariant again once it has been violated.

Because of this, the eventual program satisfies liveness, but not safety (safety is also eventually satisfied)

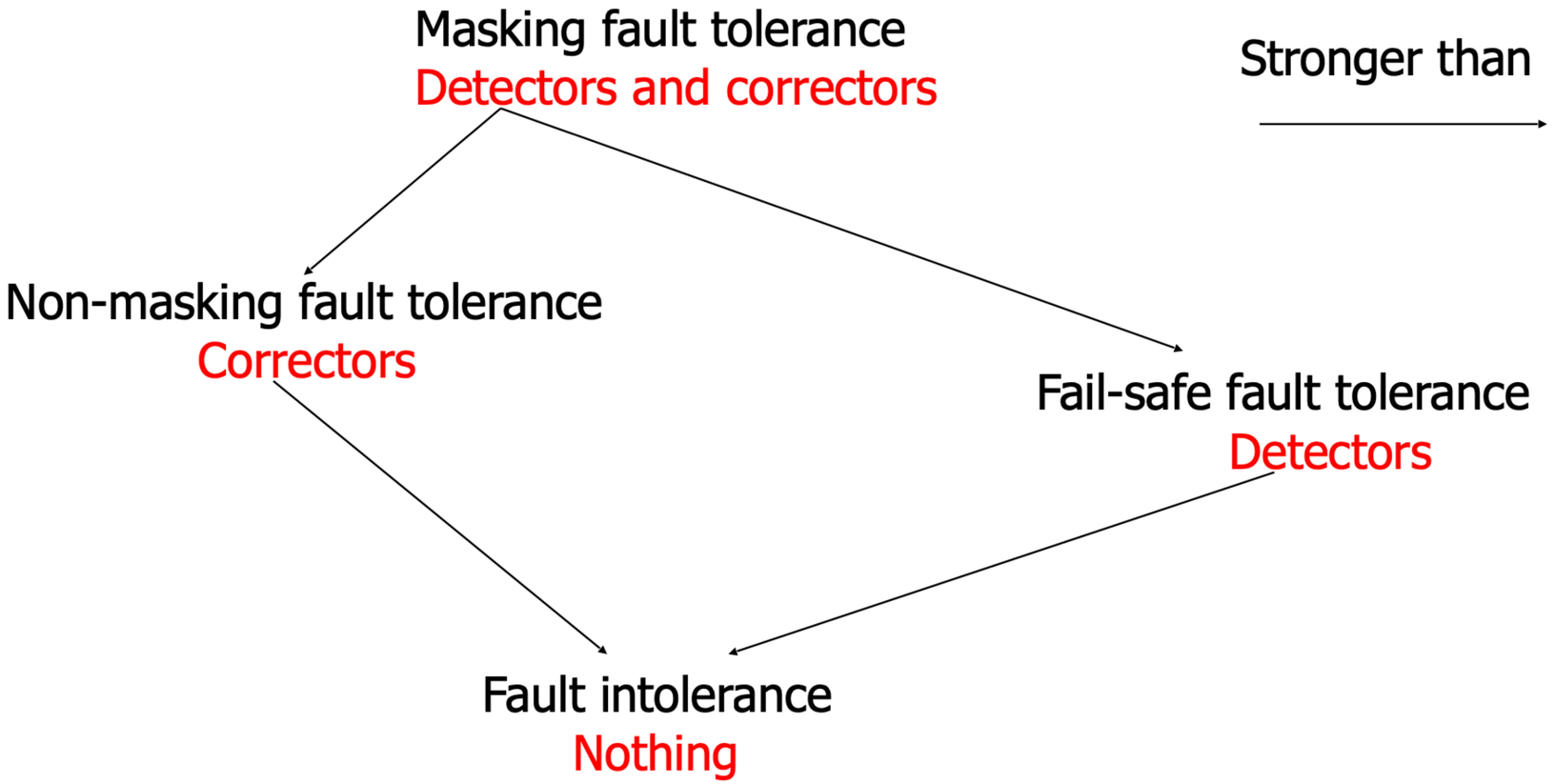
Important Results in Software Dependability

Detectors are both necessary and sufficient to design fail-safe fault tolerance

Correctors are both necessary and sufficient to design non-masking fault tolerance

Detectors and **correctors** are both necessary and sufficient to design masking fault tolerance

Designing for Fault Tolerance Classes



Phases of Fault Tolerance

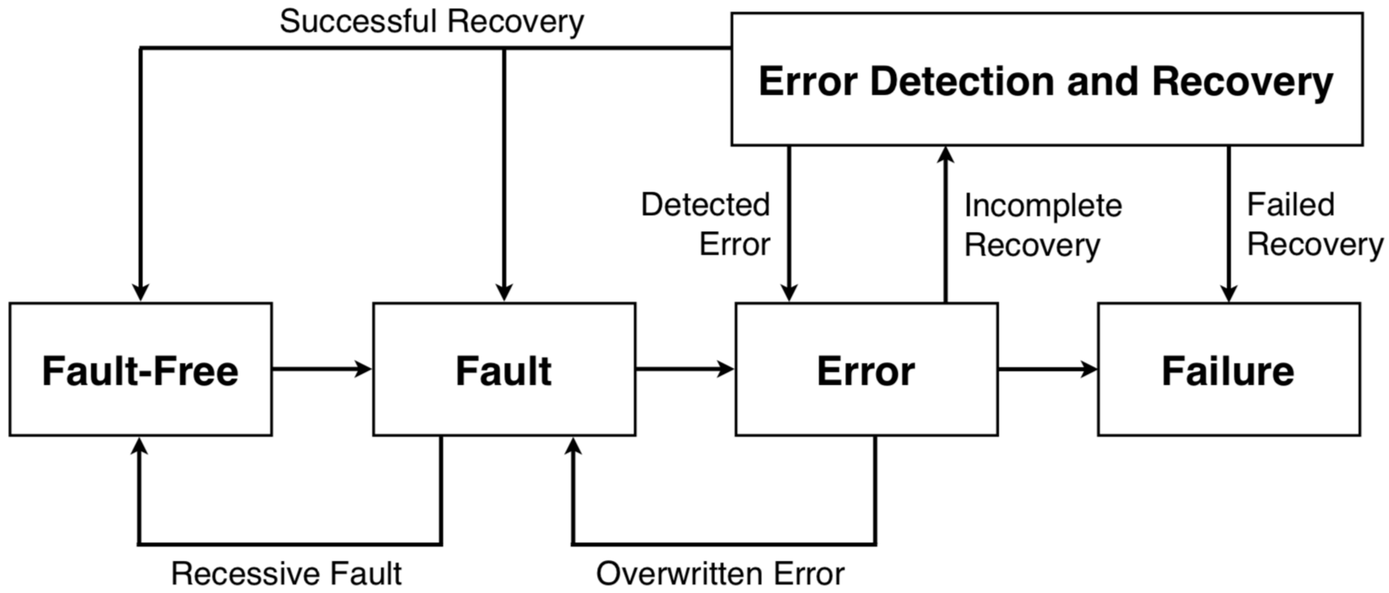
Error detection – achieved by detector components

Error recovery – achieved by corrector components

Damage assessment – investigate how far error has propagated, e.g., how many modules affected

More detectors, less damage!

Fault treatment – usually done offline to rectify problem



Fault Tolerance Techniques

There are several techniques for providing fault tolerance (fail-safe, non-masking or masking)

There are generally classified according to single module or multiple modules

Some examples of these where is it is easy to see the detection and correction elements include run-time checks and exception handlers

Detector Example - Run-time Checks

Used for detecting errors, hence used in fail-safe fault tolerance design

Many flavours:

Replication checks – compare outputs of matching modules
Timing checks – use of timers to check timing constraints
Reversal checks – reverse output and check against inputs
Coding checks – parity, Hamming codes, etc.
Reasonable checks – use semantic properties of data
Structural checks – redundancy in data structures
Validity checks – divide by 0, check array bounds, overflow

Corrector Example - Exception Handlers

Detectors raise exception (interrupt signal)

Interruption of normal operation to handle exceptional events.

Three main classes:

Interface exception – Invalid service request detected by interface detectors and corrected by service requestor

Local / internal exception – Problems with own internal operations detected by local detectors and corrected by local correctors

Failure exception – When internal errors propagate to interface, and detected by interface detectors. Global correction may be needed, e.g., look for another server

Recovering From Errors

Two main classes: **forward error recovery** and **backward error recovery**

Forward error recovery

> Upon detection of error, program attempts to get into a state which is no longer erroneous (we will see how recovery blocks enable this)

Backward error recovery

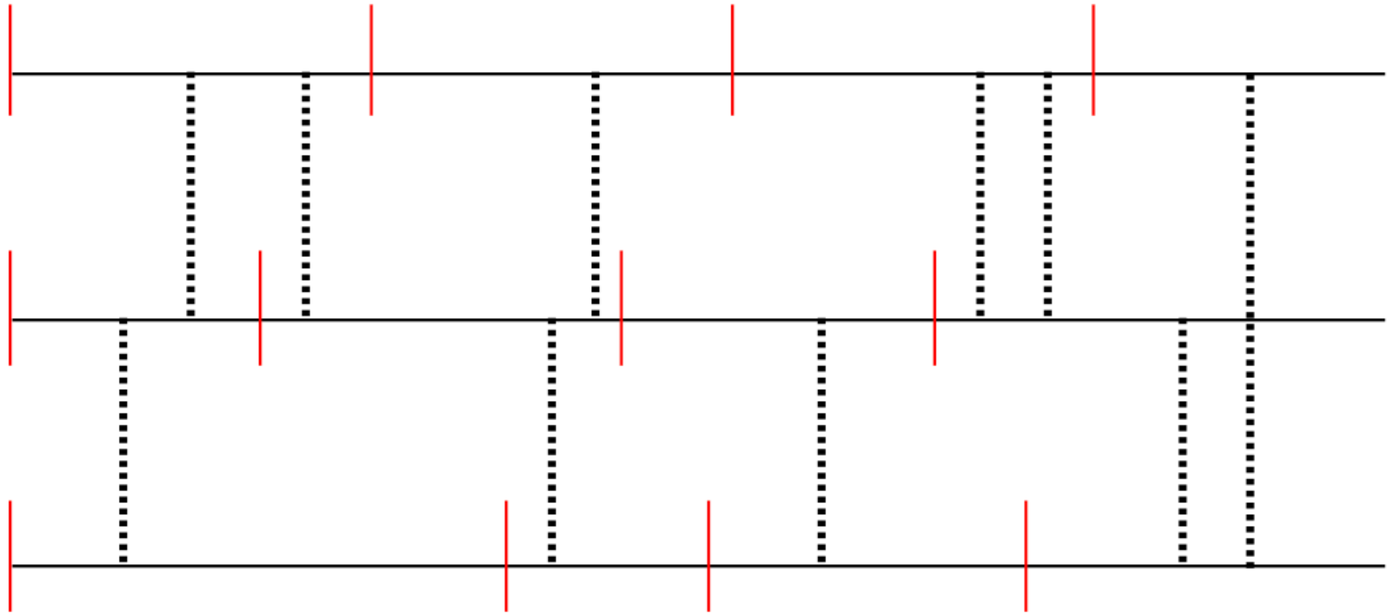
> Upon detection of error, program rolls back to a previously "recorded" good point (we will see how checkpoints enable this) from which it can restart executing

How Do We Take Checkpoints?

How can we solve problems relating to domino effects?

Checkpoints must be taken in a consistent way to avoid cascading roll-backs

We will study checkpointing approaches when we look at mechanisms for software fault tolerance

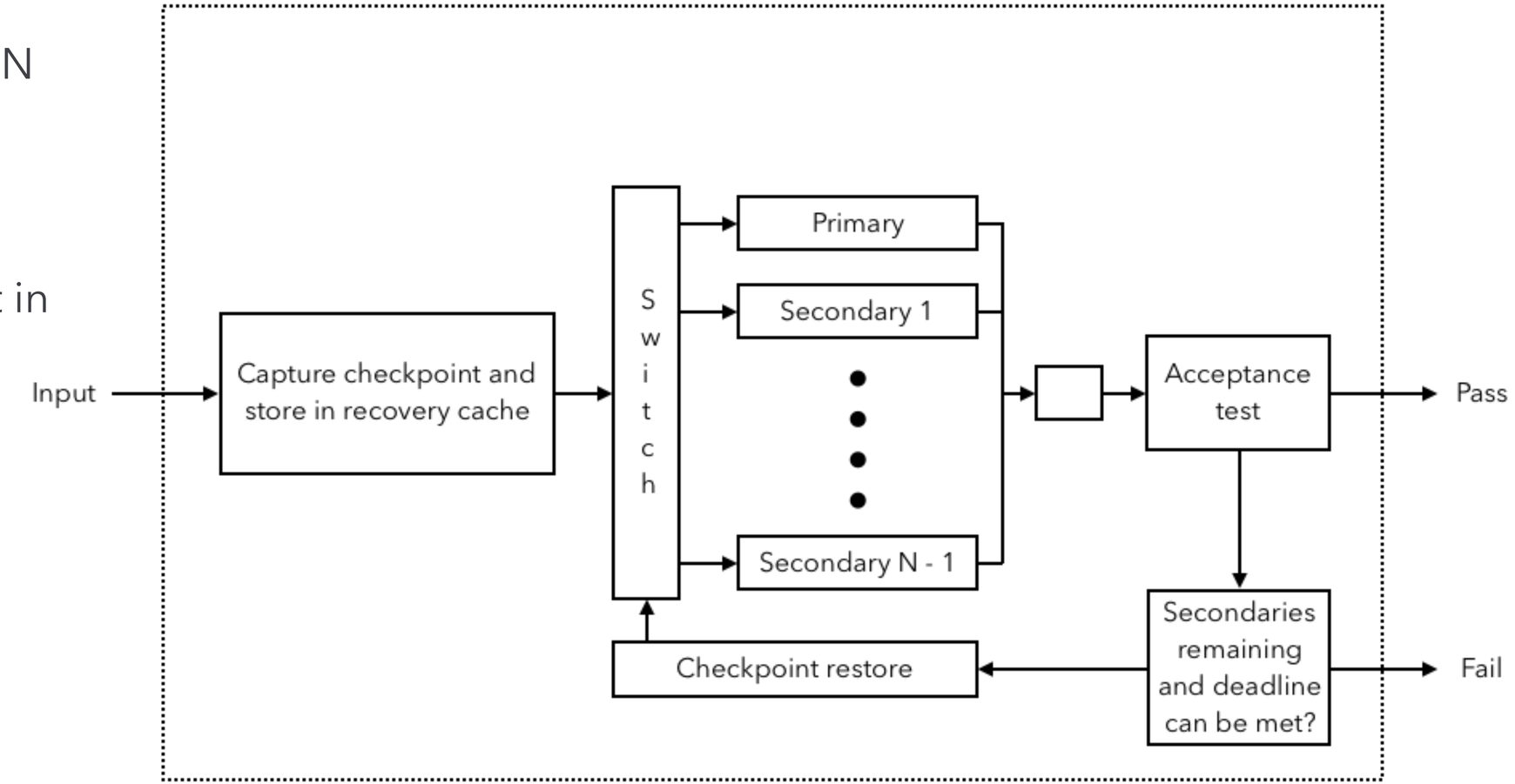


How Can We Design Recovery Blocks?

One hardware channel and N software components

Secondary components perform similar function but in a different way

We will also study recovery blocks when we look at mechanisms for providing software fault tolerance



What's Next?

We have studied:

- Basic concepts in dependability
- Fault tolerance classes
- The theory of detectors and correctors

Next we put these into action by looking at dependability analysis and evaluation