Formal Methods for Secure Systems

Computer Engineering@UniPi

Giovanni Enrico Loni

2 marzo 2024

Indice

0 Introduction

0.1 Outline of the course

1. Dependability
    - Building high reliable computer-based systems
    - Quantitative evaluation of dependability
    - Threat modeling and risk assessment
    - Malware analysis
    - Cybersecurity engineering
2. Formal methods for security
    - Formal methods applied to security
    - Case studies: Data confidentiality, Security protocols, Cyber-physical systems security

0.2 Computer-based systems

Computer-based systems are everywhere, and the services they offer are very diverses. From that, we can easy understand why the **dependability**, which is the ability of the system to deliver the expected service, is a critical aspect of these systems, in particular in a security point of view.

A system should (or must) be able to deliver the expected service, even in the presence of faults, errors, and **attacks**. This is the main goal of dependability, which is as important as the functionality of the system, pheraps even more. TO achieve that, we have **Formal Methods** that provide to us a set of techniques and tools to design, verify, and validate computer-based systems, in a rigorous and systematic way, even in presence of faults and attacks.

1 Basic concepts and terminology

All the concept and the terminology that will be presented in this section derives directly from the paper "Basic Concepts and Taxonomy of Dependable and Secure Computing" by Avizienis et al. (2004). This paper is a fundamental reference in the field of dependability and security and it is the basis for the definition of the concepts and terminology that will be used in the entire course, as suggested by the professor.

1.1 Dependability

We can give to **dependability** a simple definition: given a system, which is designed to provide a certain service, the dependability is the ability of that system to deliver the specified service also in presence of faults and malfunctions. In other words *dependability is that property of a computer-based system such that reliance can justifiably be placed on the service it delivers*. Note that the latter definition stresses the need for a justified reliance on the service, which is a key aspect of dependability.

1.1.1 Computer-based systems

A computer-based system is a system that includes a certain number of components: each of them can be interconnected and have its own functionality. The components can be hardware, software, humans and the environment in which the system operates.

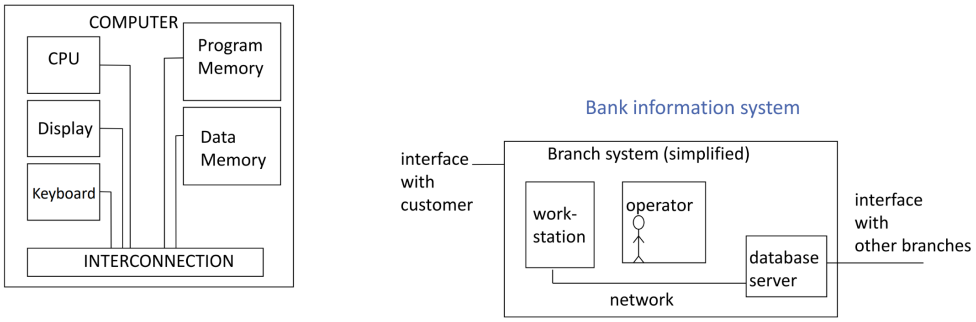

Figura 1.1: Computer-based system - C. Bernardeschi

1.1.2 Faults and Failures

We call a **failure** the inability of the system to deliver the expected service, and a **fault** the cause of that failure.

Example: if a cash machine delivers the wrong amount of money, we can say that the system has failed.

A fault causes an **error** in the state of the system, which lead to a **failure**. A failure can have different nature, such as physical, logical, human error or even as consequence of an attack.

Example: Logic Bomb. It's a piece of code that is inserted into a software system that will execute a malicious function when specified conditions are met.

```
legitimate_code();
if (date == "01/01/2020") {
    crash_system();
}
legitimate_code();
```

Computer Faults vs. Other Equipment Faults Computer faults differ from those of other equipment in several ways:

- **Subtler Failures**: computer failures are more subtle than outright crashes or sudden stops.
- **Information Storage**: computers store information in various ways, leading to a multitude of possible errors, both internally and externally.
- **Hidden Small Defects, Big Effects**: even small hidden defects can have significant impacts, especially in digital systems.
- **Complex Hierarchies**: computer systems are intricate hierarchies built upon hidden components.

1.1.3 Achieving Dependability

The dependability of a system can be achieved going trough a rigorous and engineered steps. Two main figures are involved, system and software engineers:

- **System engineers** are responsible for the design of the system, and they have to use analysis to model the dependability of the their design. From these, the software specifications are derived, and the possible changes to the system are evaluated, in order to accomodate software limitations;
- **Software engineers** are responsible for the implementation of the software, and they have to use the specifications to develop the software, and to test it in order to verify that it meets the requirements.

In general, it's crucial to understand that **dependability is not something that can be added to a system as an afterthought**. It must be considered from the very beginning of the design process, and it must be an integral part of the system, wusing a scientific and engineering approach.

1.2 The *system* entity

A simple but effective definition of a system is the following: a system is a entity that interacts with the environment and other systems; its boundaries are the common frontier between the system and the environment.

1.2.1 System's properties

A system has a **function**, which is the service that it provides to the environment, and it's described by its own functional specification. It also has a **behavior**, visualized as the sequence of states that the system goes through during its operation, and it's how the system implements its function. Then there is the **structure**, which is the way the system is organized, and it's described by its own structural specification.

From the user point of view, the system has a **delivered service**, which is the result of the interaction between the user and the system, that is the behavior of the system as perceived by the user. Obviously, the user can be seen as another system that interacts with the system under consideration.

1.2.2 System's requirements

First of all, we need to define the problem that the system has to solve, and then we have to define the requirements that the system has to meet, and then we have to define both functional and dependability requirements. Pay attention to the difference between the system's function and the system's specification: the former is the service that the system provides, while the latter is the solution implemented to provide that service. In the end, we define the **correctness** of the system, which is the ability of the system to deliver the specified service.

1.3 Dependability tree

Take in consideration the following dependability tree:

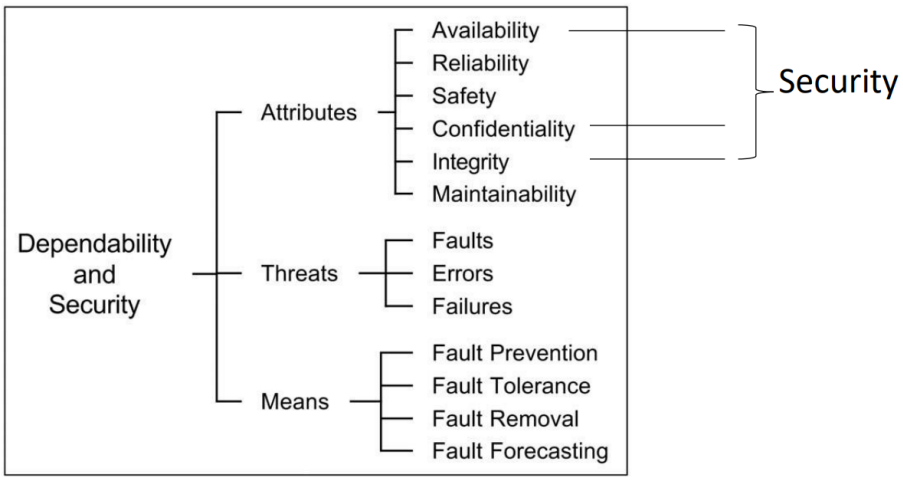

Figura 1.2: Dependability tree - Avizienis et al., 2004

1.3.1 Threats to dependability

As we said before, a **correct service** is delivered if the service is delivered in accordance with the system's specification. When this doesn't happen, we have a **service failure**, which is one the possible states of the system:

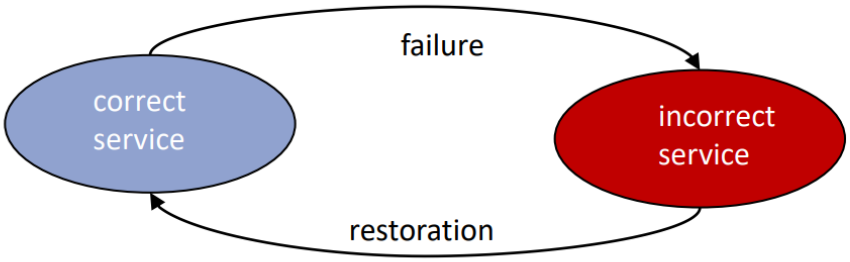

Figura 1.3: Service failure - C. Bernardeschi

We call **service outage** the period during which the system is not able to deliver the service, a **service degradation** the period during which the system delivers a service that is not in accordance with the specification, such as a subset of the services. We also recall the **chain of threats** to dependability: **faults** causes **errors**, which lead to **failures**: note that many errors don't cause failures because they don't reach the external state of the system.

Faults can be **dormant**, that is they are present in the system but they don't cause errors, and **active**, that is they cause errors, and thay can be **external** or **internal**: from the latter we can extract the definition of **vulnerability**, which is the property of the system that allows an external agent to cause a fault.

Example: Trapdoor. It's a hidden entry within a system that allows an attacker to bypass security measures.

```
username = read_username();
password = read_password();
if (username == 'dummy_user'){
    //note that the password is not checked
    grant_access();
}
if (username.isValid() && password.isValid()){
    grant_access();
}
```

From the past example we can learn that, having in mind the dependability tree, to achieve security only the authorized actions have to be allowed, and the confidentiality and the integrity of the data have appears in case of improper or unauthorized actions.

1.3.2 Dependability attributes

The dependability of a system can be described by a set of attributes, measurable and quantifiable in terms of probabilities:

- **Availability**: the readiness for correct service;
- **Reliability**: the continuity of correct service;
- **Safety**: the absence of catastrophic consequences on the user and the environment;
- **Confidentiality**: the absence of unauthorized disclosure of information;
- **Integrity**: the absence of improper system state alterations;
- **Maintainability**: the ability to undergo modifications and repairs.

We also briefly present the concept of **trust** between systems, that express the dependance of dependability of the system A from the dependability of the system B.

From them, we can gave a new definition of dependability, based on the frequency of the service failures:

> Dependability is the ability of the system to deliver the service in accordance with the specification, avoiding that service failures occur with a frequency that is greater than a certain threshold.

The threshold should be derived from the system requirements, and should consider frequency, duration and severity of the service failures.

1.4 Taxonomy of faults

1.4.1 The system life cycle

We define the **system life cycle** as the period of time that starts from the conception of the system and continues until the system is decommissioned. We're going to consider only two phases of the system life cycle: the **development phase**, during which the system is designed and implemented, and the **use phase**, during which the system is used to deliver the service.

1.4.1.1 Development phase

In this phase the system only interact with the **development environment**, such as physical world, human developers, their tools and possible facilities: in this phase development faults can be introduced in the system.

1.4.1.2 Use phase

In this phase the system interacts with the **use environment**:

- **Physical environment**: the system is subject to physical stress, such as temperature, humidity, vibrations, etc;
- **administrator**: who manages the system;
- **users**: who interact with the system;
- **providers**: who delivers the system to the users;
- **infrastructure**: everything that is needed to support the system;
- **intruders**: who try to attack the system.

The use phase alternates period of **correct delivery** of the service, **service outage** and **service shutdown**:

- **service outage**: when the system has a service failure, and it's not able to deliver the service correctly;
- **service shutdown**: when the system is stopped for maintenance or for other reasons. Note that maintenance may take place during every period of time, and includes both repair and modification.

In fact, there is a taxonomy for the maintenance: if the system is stopped for repair an active fault, then we have a **corrective maintenance**; if the system is stopped for repair a dormant fault, then we have a **preventive maintenance**; if the system is stopped for modification, then we have an **adaptive maintenance**, or **augmentative maintenance** if the system is stopped for improvement.

$

$

1.4.2 Faults classification

Given the fact that faults can't be enumerated, it's useful to classify them in order to understand their nature and their effects, because we can also identify the mechanisms that can be used to prevent that specific class of faults.

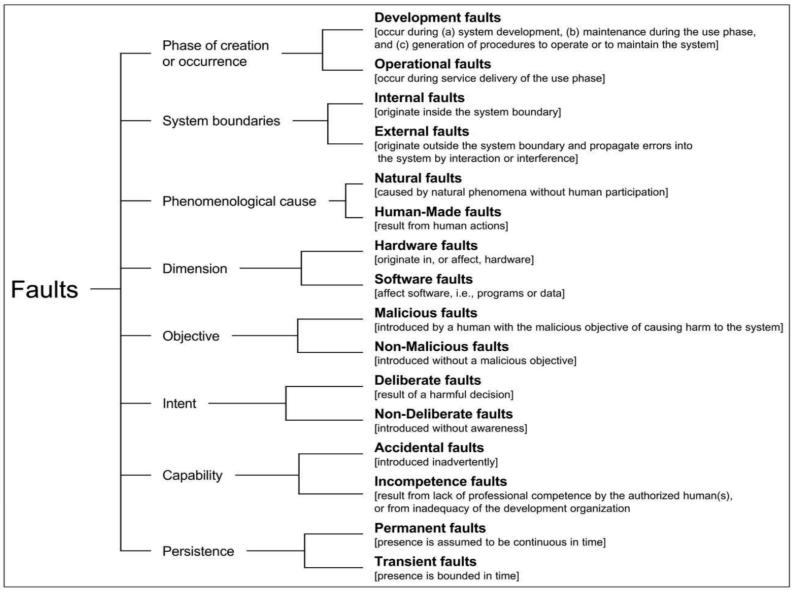

Figura 1.4: Faults classification - Avizienis et al., 2004

We identifies three main classes of faults:

- **development faults**: faults that are introduced during the development phase;
- **physical faults**: faults that affect the physical components of the system;
- **interaction faults**: faults that includes all the external faults.

Overlapping classes are possible, so we're able to identify 31 possible combinations of faults.

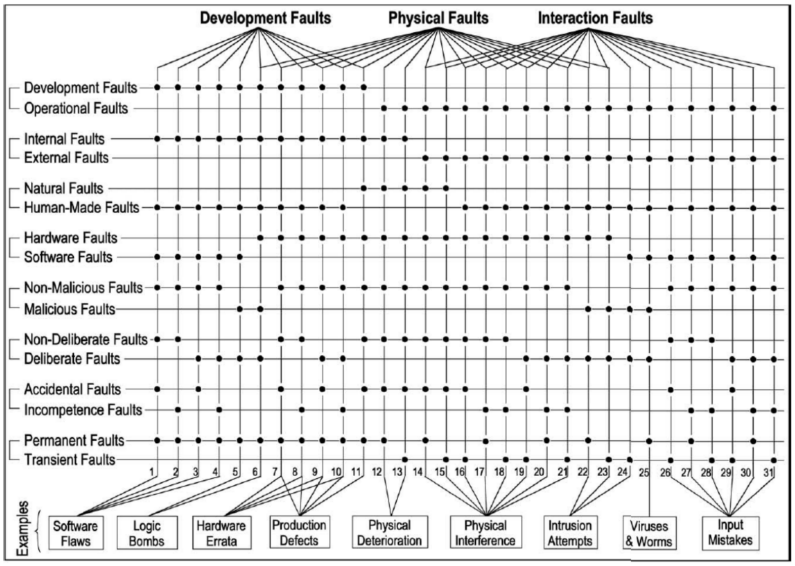

Figura 1.5: Faults classification - Avizienis et al., 2004

We're going to analyze the most important classes of faults, having in mind the previous figure.

1.4.2.1 Natural faults

They're the faults numbered from 11 to 15 in the previous figure, and they're are mainly hardware faults caused by natural phenomena, without the intervention of humans.

- **Production defect**, 11: a fault that is introduced during the development;
- **internal defect**, 12 and 13: a fault that is introduced during the use phase, and it's caused by the physical degradation of the components;
- **external defect**, 14 and 15: a fault that is introduced during the use phase, and it's caused by the physical stress of the environment, outside the system boundaries.

1.4.2.2 Human-made faults

We can distinguish between **malicious faults**, introduced with the intent to harm the system, and **non-malicious faults**, introduced without any malicious intent.

The formers have as a goal to harm the system, the latter are divided in two more classes:

- **non-deliberate faults**: faults that are introduced as human errors (1, 2, 7, 8, 16-18, 26-28);
- **deliberate faults**: faults that are introduced as a consequence of a deliberate action, such as a bad decision (3, 4, 9, 10, 19-21, 29-31).

The latter can be introduced during **development** or by **interaction**:

- **deliberate development faults**: are generally the result of a trade-off, both in terms of performance and economy (3, 4, 9, 10);
- **deliberate interaction faults**: when operational procedures are deliberately violated (19-21, 29-31).

In general, **deliberate faults** shows up only after an unacceptable behavior of the system, and it can be difficult to realize the actual faults because, when it happened, who introduced the fault can be not conscious of the consequences of his action.

However, not all mistakes and dab decisions by non-malicious humans are accidental: we can distinguish between **accidental faults** and **incompetence faults**.

1.4.2.3 Interaction faults

They can also be named as **operational faults**, given the fact that they occour during the use phase, and they're all external, because they're caused by the interaction between the system and the environment. Classes from 16 to 31 are human made, and only 14 and 15 are natural.

A common feature of these faults is the fact that they usually need the presence of a vulnerability in the system, that is a property of the system that allows an external agent to cause a fault, both intentional and unintentional.

Lastly, we recognize the **permanent faults**, that are continuous and stable, and the **transient faults**, that are temporary, even for very short periods of time.

1.5 Failures

In order to characterize the failures, we use four different dimensions, such that each of them can describe a different aspect of the failure.

1.5.1 Failure domain

This point of view leads us to distinguish between **content failures**, that are the result of the system's inability to deliver the correct service, and **timing failures**, that are the result of the system's inability to deliver the correct service at the correct time (early, late, or never).

1.5.2 Consistency domain

When a system has more than one user, it's important to understand if a failure shows up identically for each user, and we call this situation **consistent failure**, or if the failure shows up differently for each user, and we call this situation **inconsistent failure**.

1.5.3 Detectability domain

It's the property of the system to check the correctness of the service, and it's the ability of the system to detect the failure. These mechanisms have two failure modes:

- **false alarm**: the system detects a failure when there isn't;
- **missed detection**: the system doesn't detect a failure when there is.

1.5.4 Consequences domain

Consequences of a failure are divided in two classes, based on their severity and impact:

- **minor failure**: the failure has a similar cost to the benefit of the service;
- **catastrophic failure**: the failure has a cost that is much greater than the benefit of the service.

1.5.5 Criteria to evaluate the severity of a failure

We can use the following criteria to evaluate the severity of a failure:

- **availability**: the duration of the service outage;
- **safety**: possible loss of life or injury;
- **confidentiality**: possible unauthorized disclosure of information;
- **integrity**: data corruption and/or inability to recover;

1.5.6 System failures

When a system has a failure, it is usually caused by different coexisting faults; we talk about **single fails** if the failure is caused by a single fault, and **multiple fails** if the failure is caused by multiple faults. In the latter we can also divide the faults in **independent** and **dependent**, respectively if the faults are not related to each other, and if the faults have a common cause.

1.5.7 Dependability and security failures

These failures occurs if the system suffers service failures more frequently than an acceptable threshold; even the specifications of the system can contain faults:

- **omission**: a requirement is not included in the specification;
- **unjustified requirements**: choice of requirements that are not justified by the system's function, that raises the cost of the system.

Systems can have different type of failures:

- **fail-controlled**: the system is designed to fail in a controlled way, described by the system's specification;
- **fail-stopped**: the system in which possible failures are only *haltings*;
- **fail-silent**: the system in which possible failures are only *silents*;
- **fail-safe**: the system in which there are only minor failures;

1.6 Errors

As we saw, an error is a part of the system state that is able to lead to a failure. An error is detected if a signal to indicate its presence in raised, otherwise the error is undetected and it's called **latent error**. It's not true that every error leads to a failure, and this depends on:

- the **structure of the system**, specially the presence of redundancy;
- the **behavior of the system**, for example the part of the state that contains the error is never reached during the delivery of the service, then the error is not able to lead to a failure.

The classification of the error is done according to the damage pattern (single, double or triple bit, burst, etc.), and how many components are affected by the error (single, multiple, etc.).

1.7 Chain of threats - Relationship between faults, errors and failures

When systems have to interact with each other, an error propagation can occur:

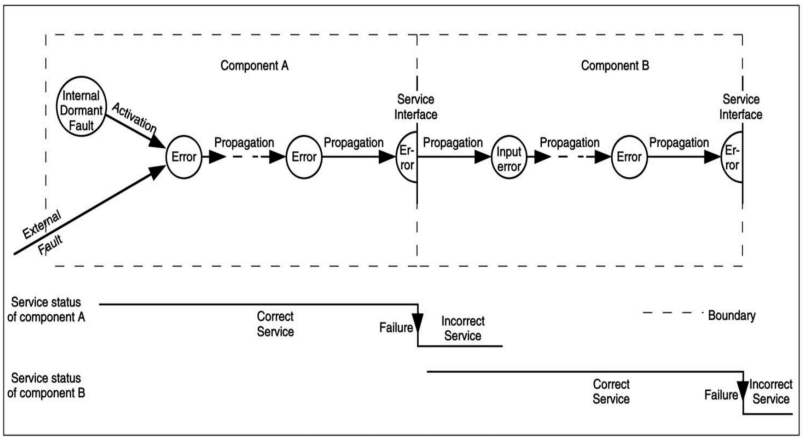

Figura 1.6: Chain of threats - Avizienis et al., 2004

An **active fault** is when it leads to an error, both if it's internal or external, and it can lead to a failure, and a **fault activation** occurs when a particular input activates a dormant fault in a specific component. This can also be a **propagation** between the components, where an error of a component can lead to an error of another component.

Example: error propagation

> Take a sensor which reports the spinning speed of a turbine. If the sensor fails and starts to report that the turbine is no longer spinning, it inject incorrect data (fault) into the control system, that will send to the turbine the wrong commands, and the turbine could be damaged (failure).

1.8 Dependability means

When we talk about dependability means, we refer to the mechanisms that are used to prevent, detect and tolerate, or in general deal, with faults.

1.8.1 Fault prevention

These techniques are usually related to general system design, and they're used to avoid the introduction of faults in the system. They can be divided in two classes:

- **development fault prevention**: techniques that are used to avoid the introduction of faults during the development phase;
- **improved development process**: techniques that are used to improve the development process, such as the use of formal methods, the use of a rigorous testing and so on.

1.8.2 Fault tolerance

It's the ability to deal with faults at run-time, and ensure that the system is able to deliver the service even in presence of faults. There are a lot of techniques that can be used to achieve fault tolerance, as we can see in the following figure:

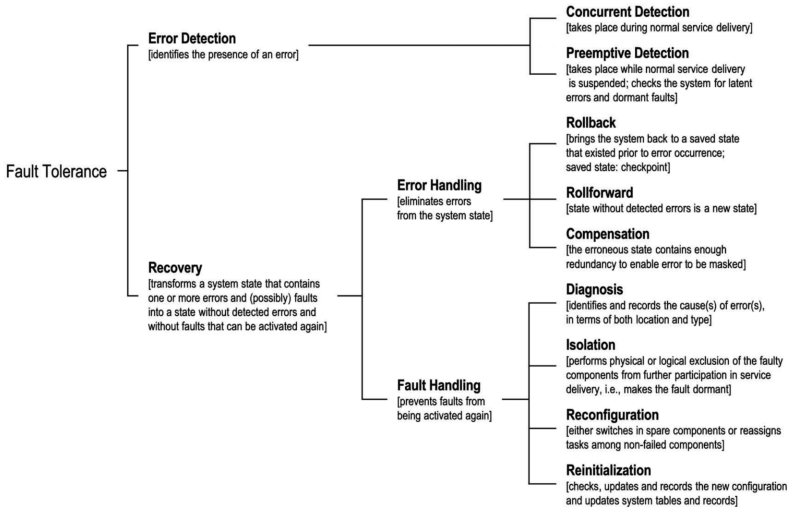

Figura 1.7: Fault tolerance techniques - Avizienis et al., 2004

1.8.2.1 Example of error detection

For simplicity, let's state that *when the error reaches the boundaries of the systems, then we have a failure*. In this context, the most challenging aspect are:

- the **identification of the error**, that is the ability to detect the presence of the error;
- **ensure that status containing the error is never reached**, that is the ability to avoid that the error leads to a failure;
- the **prevention of the error propagation**, that is the ability to avoid that the error of a component leads to the error of another component.

Example: error detection with two systems

> Take two systems, A and B, that should provide the same service. If they get the same input, then they should provide the same output. If the outputs are different, then we have a detection of the error, within the hypothesis that the systems are independent and it's very unlikely that they have the same error at the same time.

1.8.3 Fault handling

When we talk about fault handling, we refer to the mechanisms that **prevents faults from being activated again**. It's composed by different phases:

1. **diagnois**: the phase in which the system detects the presence of the fault. Usually, a component is made in order to test another components, and the aim is to identify and records the cause of the error, in terms of location and type;
2. **isolation**: to obtain the physical and/or logical exclusion of the faulty component from the rest of the system;
3. **reconfiguration**, such as thw switch to a redundant component, or the use of a different path to reach the same component;
4. **reinitialization**: to restore the system to update the system to the new configuration.

In conclusion we state that the **system recover is composed by the error handling phase and the fault handling phase**.

1.8.4 Fault removal

The fault removal is the process that is used to remove the faults from the system, and it's usually done during the development phase. The main goal is to remove the faults that are present in the system, and to prevent the introduction of new faults. It's composed by different phases, that we'll see in the next sections:

1.8.4.1 Verification phase

The verification phase is the phase in which the system is tested to verify that it meets the **verification conditions**. To do that, there are two main methods:

- verification **without execution**: the system is tested without executing it, and it's usually done via inspection or theory-proving. A state-transition diagram can be used to verify the correctness of the system, and it's applicable to various type of the system, and applicable to fault tolerance mechanisms. Worth to mention the fact that, in this type of verification errors and faults are artificially injected as part of the test pattern;
- verification **by execution**: the system is tested by executing it, and it's usually done via **dynamic verification** (e.g. symbolic execution, testing both for hardware and software), **deterministic testing** and **statistical testing**.

Another two steps are crucial:

- **verification of the mechanism**: the verification of the fault tolerance mechanism, and it's usually done via **fault injection**;
- **verification of the system**: ensure that the system cannot do more than what is supposed to do, and it's usually done via **penetration testing**.

To remove a fault during the exercise, both **corrective maintenance** and **preventive maintenance**.

1.8.5 Fault forecasting

The fault forecasting is done by performing an evaluation of the system behavior, with respect to fault occurrence and activation, and it's usually done via **qualitative evaluation** and **quantitative evaluation**.

1.8.6 Error recovery

1.8.6.1 Error compensation

When we talk about fault tolerance, we're in fact talking about **fault masking**: a general method to achieve this goal is **performing multiple computations** trough replicas, and then apply a vote mechanism to the results. It's worth to remember that hardware faults **fails independently** and, on the contrary, software faults **fails dependently**: to achieve a sort of independence, we can use **design diversity**, that is the use of different design techniques to implement the same function.

Example: error compensation with TMR

> Take a system that uses a **Triple Modular Redundancy** (TMR) to achieve fault tolerance. The system has three replicas, and the output is the result of a majority vote. If one of the replicas fails, then the output is still correct, because the other two replicas are still able to provide the correct output.

1.8.6.2 Organization of fault tolerance

We can summarize the organization of fault tolerance in the following figure:

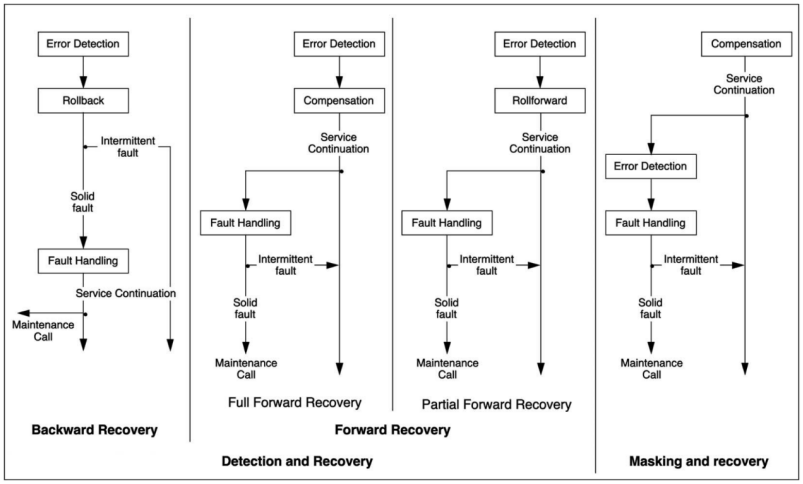

Figura 1.8: Organization of fault tolerance - Avizienis et al., 2004

WE just have to add some definitions:

- **solid faults** are those that are permanent and their activation is repeatable;
- **elusive faults** are those that are permanent and their activation is not systematically reproducible;
- **intermittent faults** are those with transient physical or interaction faults, and their activation is not systematically reproducible.

Remember that the classes of faults that can be actually tolerated depend on the fault assumption that is being considered in the development process, and on the independence of the redundant components that are used to achieve fault tolerance.

1.9 Error detection

The error detection is the ability of the system to detect the presence of an error, and different strategies can be used to achieve this goal, such as:

- **replication checks**: the use of multiple replicas to perform the same computation, and then compare the results, under the assumption that the replicas fail independently;
- **reasonability checks**: the use of a model of the system to check the reasonability of the output, and then compare the output with the model;
- **run-time checks**: mechanisms provided via hardware or software, like division by zero, array bounds, etc.

- **specification-based checks**: the use of the problem specification to check the correctness of the output (e.g. to find a solution to an equation, we can substitute the result in the equation and check if the result is correct);
- **reversal checks**: the use of the inverse function to check the correctness of the output.
- **structural checks**: the use of known properties of the system to check the correctness of the output.
- **timing checks**: the use of watchdogs to check the timing of the system.
- **codes**: the use of codes to check the correctness of the output (e.g. parity, checksum, etc.).

1.9.1 Structural approach to error detection

The main goal is to prevent the propagation of the error, and to achieve that some structural properties should be set to help the system.

1.9.1.1 Principle of least privilege

the concept of **minimum privilege** is crucial, and it's the idea that a component should have the minimum privilege to perform its function, and nothing more. Following this idea, we should consider the fact that **no action is permissible unless it is explicitly allowed**, also known as the concept of **mutual suspicion**.

1.9.1.2 System modularization and partitioning

Remembering the fact that a system should be modularized, we can use the **modularization** to prevent the propagation of the error, adding to each module an error detection (and possibly recovery) mechanism, in order to confine the error to the module in which it occurred and don't let it spread to the other modules. The last reasoning is also valid for the **partitioning** of the system, when modules act independently and the error can't spread to the other modules.

1.9.1.3 Temporal structuring

Another thing to take in consideration is the **temporal structuring** of the activities between the modules, for those operations only between two specific modules that don't communicate with the rest of the system. We also introduce the concept of **atomic action**, that is an action that is performed in a single step, and it's not possible to interrupt it: if a failure occurs, only the participating actions are affected.

1.9.2 Measurement of effectiveness of error detection

Different metrics can be used to measure the effectiveness of the error detection, such as:

- **coverage**: the probability that an error is detected, given that it actually occurs;
- **latency**: the time that elapses between the occurrence of the error and its detection;
- **damage confinement**: the probability that the error is confined to the component in which it occurred;
- **forward recovery**: the probability that the system is able to recover from the error, transforming the erroneous state into a **new** correct state;
- **backward recovery**: the probability that the system is able to recover from the error, transforming the erroneous state into the **previous** correct state.

It's worth to spent some words for the last two metrics, that will be discussed in the next section.

1.9.2.1 Forward recovery

This technique requires to **asses the damage cause** by the detected error **propagates before detection**, and it's usually implemented ad-hoc for the specific system. An effective **example** is the following:

> In a real time control system, a situation when input a sensor input is occasionally missed is tolerable, and the system should implement a forward recovery by skipping its response of the missed input.

1.9.2.2 Backward recovery

This technique is a little bit more complex, because **requires a previous correct state** to be restored, also called **checkpoint**, and can be tedious, especially in case when multiple modules are involved, because we need to restore a **consistent checkpoint** for each of them, as we can see in the following figure:

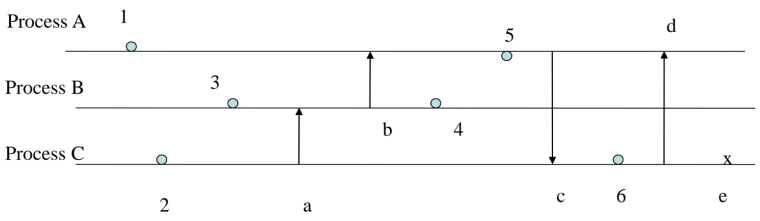

Figura 1.9: Backward recovery - C. Bernardeschi

In this image we see the checkpoint, as circle, the passed messages between the modules, and the error that occurs with a X: to avoid a **domino effect**, we need to restore a consistent checkpoint for each module, also considering their communications, remembering the concept of atomic action.

The basic issues of backward recovery are:

- loss of computation time between the checkpoint and the rollback;
- loss of data between the checkpoint and the rollback;
- the need of a specific mechanism that implements the rollback;
- the increase of the overhead of the system, in order to restore the correct state.

The class of faults that gain benefits from the backward recovery are the **transient faults**, because they usually disappear after a short period of time, in **parallel computing**, to avoid a complete restart of the system, and in **real-time systems**, to avoid the loss of the real-time constraints.

On the other hand, the class of faults that are not suitable for the backward recovery are the **hardware and design faults**, because the system will always do the same action, resulting in the same error.

1.9.3 The exception handling

The exception handling is a mechanism that is used to deal with the errors, and it's usually implemented via software, and it's used to deal with the errors that are detected at run-time. The main goal of the exception handling is to avoid the propagation of the error, and to restore the system to a consistent state. Three are the main classes of exceptions:

- **interface exceptions**: exceptions that are raised when the system receives an input that is not in accordance with the specification, handled by the module that requests the service;
- **internal local exceptions**: exceptions that are raised when the system detects an error in its own state, handled by the module itself;
- **failure exceptions**: exceptions that are not handled by the mechanism, communicated to the user.