Composition with Guarantees for High-integrity Embedded Software Components Assembly

Project Number 216682

D3.3 – Mechanisms and components for execution platforms supporting dependability

Version 1.0

24 January 2011

Final

Public Distribution

CNR-ISTI, Atego, Aicas, MDH, FhG, TCF, FZI, EAB, Atos Origin

Project Partners: **Aicas, Atego, Atos Origin, CNRI-ISTI, Enea, Ericsson, Fraunhofer, FZI, GMV Aerospace & Defence, INRIA, Intecs, Italcertifier, Maelardalens University, Thales Alenia Space, Thales Communications, The Open Group, University of Padova , University Polytechnic of Madrid**

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial version	2010-05-30
0.2	New Draft from Atego	2010-09-23
0.3	Taking into account Partner's inputs	2010-10-26
0.4	Restructuring section 2 concerning threats classification	2010-11-03
0.5	Taking into account Partner's refinements about threats and Mitigation Means description	2010-11-19
0.6	- Add a mean to uniquely identify each threat - Enrich threats lists according to Partner's inputs - Add Mitigation Means coverage table for each Mitigation means - Update some Mitigation Means description according to Partner's inputs.	2010-12-10
0.7	- Integrate finalized Inputs from Partners - Add global coverage tables	2010-12-21
0.8	Atego internal review	2011-01-06
1.0	CHESS Internal review	2011-01-24

TABLE OF CONTENTS

EXECUTIVE SUMMARY

The purpose of the document is to identify runtime mechanisms and components for execution platforms that can guarantee dependable and secure executions of the applications running on them. The approach that has been taken is to identify assets from a definition of an Execution Platform common to the domains addressed in CHESS in order to elaborate a list of threats (faults, errors and failures) that could potentially compromise these assets. Once the threats identified, mitigation means aiming to counter these threats will be identified.

The mitigation means are grouped into three major categories:

- **Fault prevention**: means to prevent the occurrence or introduction of faults.
- **Fault tolerance**: means to avoid service failures in the presence of faults.
- **Fault removal**: means to reduce the number and severity of faults.
- **Fault forecasting**: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and dependability specifications are adequate and that the system is likely to meet them.

Coverage tables are then established to ensure that each identified mitigation mean addresses (mitigated or prevented) at least one of the threats identified and that each threat is addressed (mitigated or prevented) by at least one of the identified mitigation means.

GLOSSARY

Assets: Part or feature of the system that has value and that should be protected against threats.

Failures: Event that occurs when the delivered service deviates from correct service.

Errors: Part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the provided service.

Faults: Adjudged or hypothesized cause of an error.

Threats: Faults, errors and failures that may affect the system, preventing it from attaining the required dependability properties.

Mitigation means: Mechanisms, algorithms, protocols and extensions for execution platforms useful to support dependability and guarantee to the applications the achievement of a given level of dependability.

1. INTRODUCTION

The goal of this document is to define mechanisms, algorithms, protocols and extensions for the execution platforms useful to support dependability and guarantee to the applications the achievement of a given level of dependability and trustworthiness. The solutions of interest might well include runtime error detection mechanisms and facilities based on continuous monitoring of system/application conditions and fusion of the gathered information according to re-defined rules. The final goal is to detect unwanted events such as the occurrence of faults or changes in the environment or working conditions (possibly beyond acceptable thresholds) and to understand/diagnose their root cause. Mechanisms supporting application recovery and general reconfiguration policies are also included. These will serve against accidental faults, but also make the system more resistant and will take both the form of pro-active actions and as adaptation/correction actions in response to changes in the system conditions, bugs, and possibly user needs.

1.1 DESCRIPTION OF THE PROBLEM

To identify and specify which mitigation means the execution platform should or shall carry out, we must first define the execution platform itself. In a second step, a list of threats susceptible to compromise the dependability of the Execution Platforms considered in the specific industrial domains targeted by CHESS, will be established. Finally, mitigation means, and particularly runtime mechanisms, able to counter these threats, in order to guarantee the dependability of the Execution Platform itself and the applications running on it, will be identified.

1.1.1 Security concerns

The starting point for WP3 (and WP4) activities was the set of non-functional requirements identified by the industrial CHESS partners and summarized in D1.1. Focusing on dependability and security requirements, it turned out that the large majority of the identified non-functional requirements were related to the dependability, while security aspects were basically neglected.

Even the only two non-functional requirements labeled as pertaining to security (see D1.1, page 96, ID 127 and 128) actually concern dependability aspects, the first requiring the capability to evaluate fault tolerance structures and the second the capability to evaluate safety level induced by recovery mechanisms. Other requirements (ID140 and ID198) only mention security in generic terms.

Therefore, despite the original plan was to address both dependability and security aspects within WP3, the effort has been fully devoted to address the dependability concerns

1.2 EXECUTION PLATFORM

This section aims to define Execution Platform taking into account the wide-variety of execution platforms used in the industry domains targeted by CHESS.

1.2.1 Definition

A definition of an Execution Platform could be:

From a functional point of view, an Execution Environment is a set of hardware and software components providing services (e.g. Computing, Memory Management, input/output, etc.), necessary to support Applications.

1.2.2 Core Components

An Execution Platform (EP) consists of the following elements:

- One or more processing unit that:
  - initializes the EP during boot
  - defines the EP Instruction Set Architecture (ISA)
  - is used for executing code built using this set of instructions
- A set of connections (e.g. buses) between the processing unit and code/instructions, data, resources (I/O)
- Physical Memory for storing data and code
- A mechanism to initialize the boot process
- The code and data of the Execution Platform itself

1.2.3 Services

An Execution Platform makes available its services through an interface in the form of an Application Programming Interface (APIs) or an Instruction Set Architecture (ISA), depending on the type and implementation of the interface.
Here is a non-exhaustive list of the kind of services that an Execution Platform may provide:

- **Execution Management**: dynamically manage the execution of code (e.g. thread execution, interrupt handling, scheduling, synchronization mechanisms, application loading/unloading) and whether the code is executed natively or interpreted (e.g. byte code). Execution management may provide several execution options (e.g. normal, debug, monitor).

- **Memory Management**: allocate/deallocate, map/unmap, protect/unprotect, and lock/unlock memory areas, either from anonymous main memory or from specific physical areas to access specific I/O and/or communication resources.

- **Storage & File Management**: enable applications to store and retrieve data in non-volatile storage. Such storage service could also be used to store applications themselves, to avoid them being resident in main memory.

- **Intra-EP Communications**: enable applications running within the EP to communicate between themselves. An operating system would typically provide IPC (Inter-Process Communication) mechanisms to its applications (processes).

- **Application Management**: configure, install, upgrade, and manage applications.

- **Network & Communication**: enable applications to establish connections with the external world through whatever protocol stack is relevant to the applications and the EP.

- **Input/output**: access I/O devices either in some raw form or through more elaborate services.

- **Credential Management**: identify users and processes, and assign them credentials and permissions so that Access Control policies may be enforced.

Services may vary from one Execution Platform to another. Similar services may be offered in different ways (different abstractions and APIs) by different EPs. However, Execution Platforms identified for the CHESS project may implement all or a subset of the described individual services.

1.2.4 Multiple Execution Platform services

More than one EP can be present on the same hardware. In this case, a resource can be:

- **Dedicated**: the resource is dedicated to a particular EP

- **Shared**: the resource is shared between multiple EPs

Some resources share assets with one or multiple EPs: for example they may share the access to a given bus. These resources are an extension of the EPs, and must be considered together as a whole.

When multiple EPs are present on the same hardware there must be ways to configure the set of resources that are owned by an EP as well as the set of resources shared with other EPs. Such configurations may be pre-defined and can be static or dynamic.

Resources granted to an EP should be protected from faults from other EPs.
An multi-EP may offer the following services:

- **Configuration Management**: Following defined security policies, this service permits the ability to dynamically:
    - Create/destroy/monitor/debug an EP
    - Allocate a set of dedicated or shared resources available to that EP either at start time or during run-time
    - De-allocate dedicated or shared resources previously allocated to an EP. Note that upon destruction of an EP all resources previously allocated to that EP are de-allocated (recycled)
    - Switch resources to simplify dynamic management of resources so that they can be easily de-allocated from an EP and allocated to another one, through a single “switch” operation
    - Account for EP resource usage
    - Manage (define, delete, modify) the Dependability Policy applicable to the set of resources owned by an EP.

- **Inter-EP Communications**: EPs may offer to their applications a facility to establish communication channels. Such channels can be used to exchange data between applications running in different EPs. Communication channels may also be destroyed. Inter-EP communications channels could include the use of shared resources such as memories.

2. DEPENDABILITY ANALYSIS FOR EXECUTION PLATFORM

This section aims to globally list assets of an Execution Platform and the related threats in order to identify the mitigation means to enforce as countermeasures of these identified threats.
One could then select the mitigation means to enforce on one's own execution platform, according to threats one wants to address.

2.1 ASSETS

The list of assets of an Execution Platform, established for the CHESS project, to protect against threats is the following:

- **Processing Unit:** the processing unit is used for executing code built using its set of instructions. The correct functioning of the Execution Platform completely depends on the accurate functioning of the CPU.

- **Memory:** Memory devices (RAM, Flash, ROM, SRAM) are used for storing code and data. A corruption or unintentional disclosure of information could alter the correct functioning of the whole system (application and underlying Execution Platform)

- **Communication:** Communication channel could be established between two Execution Platforms. It is so important to ensure the availability of these channels as well as the integrity and confidentiality of the data transiting through them.

- **Device**: Some devices provide services to critical application and are required to be protected against certain types of attacks or faults such as denial of services or unauthorized modifications of their registers.

- **Execution Platform code and data**: The Execution Platform needs to be itself protected against attacks and faults that are likely to corrupt data or code in order to avoid propagation of errors to the rest of the system.
  For instance, the configuration parameters allow the integrator to manage the rights of every application on the system. It is essential that these data not be corrupted in order to avoid any attacks or faults that are likely to escalate the rights of a particular application

- **Real time**: Real Time capabilities such as interrupt latency must be preserved to ensure the correct functioning of Real Time applications. An increase of interrupt latency caused by a specific attack or fault may involve a disturbance in the functioning of the application, possibly leading to
- denial of services.

2.2 THREATS

This section aims to list the possible threats that could compromise the assets previously identified.

The threats to dependability are faults, errors and failures that may affect the system, preventing it from attaining the required dependability properties. A systematic description and classification of threats is provided in [3], and it is summarized in the following. A system delivers correct service when the service implements the system function. A **failure** is an event that occurs when the delivered service deviates from correct service. An **error** is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the provided service. A **fault** is the adjudged or hypothesized cause of an error.

There are causality relationships between faults, errors, and failures. Faults are activated by the application of an input to a component that causes a dormant fault to become active. By propagation, several errors can be generated before a failure occurs. Error propagation within a given component (i.e., internal propagation) is caused by the computation process that transforms errors into other errors. A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. Error propagation from component A to component B that receives service from A (i.e., external propagation) occurs when, through internal propagation, an error reaches the service interface of component A. At this time, the service failure of A appears as an external fault to B and propagates the error into B via its used interface.

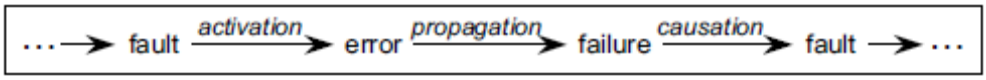


Figure 1: The fundamental chain of dependability threats

2.2.1 Faults

The following table identifies the most relevant faults associated with each Asset for the domains targeted by the CHESS project. The faults have been specified in collaboration with the industrial user partners in the project reflecting those that are most common or of most concern.

Faults	Description	Domain*
Processing Unit		
FLT-CPU-1	Address-decoder fault (AF): No cell is accessed by certain address, multiple cells are accessed by certain address, certain cell is not accessed by any address, or certain cell is accessed by multiple addresses.	R
FLT-CPU-2	Faults in the instruction decoding and control function. The faulty behavior of a CPU can be specified as follows. When instruction I_X is executed any one of the following can happen: • Instead of instruction I_X some other instruction I_Y is executed. • In addition to instruction I_X, some other instruction I_Y is also activated. • No instruction is executed (for example, this happens in case of hang or crash of the CPU).	R, A

FLT-CPU-3	Faults in the data transfer function. Under a fault in the data transfer function for any instruction I_X: • A line in a transfer path can be stuck at 0 or 1. • Two lines of a transfer path can be coupled, i.e. they fail to carry different logic values (for example, this can happen due to metallization shorts or capacitive couplings).	R
FLT-CPU-4	Faults in the data manipulation function. The data manipulation function refers to various functional units such as the Arithmetic Logic Unit (ALU), interrupt handling hardware, hardware used for incrementing (or decrementing) the stack pointer, index register or program counter, hardware used for computing the addresses of operands in various addressing modes such as indexed and relative, etc.	R, A
FLT-CPU-5	Faults in the register decoding function. Register decoding refers to the task of decoding the address of a register which may be stored as a specific bit pattern in the instructions involving that register or which may be generated by the control unit during the execution of the instructions. Possible faults may lead to the following situations: • Whenever a register R_X is to be accessed (while executing any instruction which involves R_X), no register is accessed, i.e. when R_X is to be written, its contents remain unchanged and when the contents of R_X are to be retrieved, no content is retrieved (the bit of vector which should contain the content of the register are all set to one or to zero, depending on the technology in use). • Whenever R_X is accessed, all registers are accessed. Whenever R_X is to be written with data, all the registers will be written with data, and whenever the contents of T_X are to be retrieved, the contents formed by the OR or AND function (depending on the technology in use) over the all registers will be retrieved.	R
FLT-CPU-6	Faults in the data storage function. A fault model for the data storage function accounts for the faults in various registers: any cell of a register can be stuck at 0 or 1, and this fault can occur with any number of cells and with any number of registers.	R
Memory		
FLT-MEM-1	Stuck-at fault (SAF): Cell (line) stuck-at 0 or stuck-at 1.	R, A
FLT-MEM-2	Transition fault (TF): Cell fails to transit from 0 to 1 or from 1 to 0.	R
FLT-MEM-3	Bridging fault (BF): Short between cells (AND type or OR type).	R
FLT-MEM-4	Stuck-open fault (SoF): Cell is not accessible due to broken line	R
RAM		
FLT-MEM-5	Coupling fault (CF): The coupling fault is static, if coupled (victim) cell is forced to 0 or 1 if the coupling (aggressor) cell is in given state. It is an inversion coupling fault, if transition in coupling cell complements (inverts) coupled cell. It is an idempotent coupling	R, A

	fault, if coupled cell is forced to 0 or 1 if coupling cell transits from 0 to 1 or 1 to 0. There are also intra-word and inter-word coupling faults distinguished in word-oriented memories	
FLT-MEM-6	Neighborhood pattern sensitive fault (NPSF): A pattern sensitive fault is a general (multi-cell) coupling fault, which causes the content of a memory cell, or the ability to change the content, to be influenced by certain patterns of other cells in the memory. The neighborhood pattern sensitive fault (NPSF) means that the aggressor cells are the neighborhood of the victim cell (5 or 9 neighbors). Static (forcing a certain state of the victim), active (forcing a certain transition of the victim), and passive (the victim cannot change its state) pattern sensitive faults can be distinguished.	R, A
Flash		
FLT-MEM-7	Word-line program disturbance fault (WPDF): a cell transits from 1 to 0 when another in the same word-line is being programmed (1 to 0).	R
FLT-MEM-8	Word-line erase disturbance fault (WEDF): a cell transits from 0 to 1 when another in the same word-line is being programmed (1 to 0).	R
Communication		
FLT-COM-1	Interaction Fault (External fault) - Line transmission disturbance	T, A
FLT-COM-2	Interaction Fault (External fault) - Signaling link down	T, A
FLT-COM-3	Interaction Fault (External fault) - Transmission path blocked	T, A
Device		
FLT-DEV-1	No or continuous DMA (Direct Memory Access) access	R
Execution Platform code and data		
FLT-EP-1	Development Fault - Misunderstood specification or classic coding bugs in the implementation of the state-transition machine, whose states correspond to the operation modes and transitions represent the possible transitions between operation modes (FA01). It may result in incorrect initialization or invalid trajectory in the system state space.	R, S
FLT-EP-2	Development Fault - Faults in the implementation of the scheduling algorithm (FA02), which may result in incorrect order of task invocations or synchronization issues, as well as load balancing issues.	R, A,T, S
FLT-EP-3	Damage of registers or memory storing synchronization variables (FA03), which may result in incorrect order of task invocations or synchronization issues.	R, S
FLT-EP-4	Damage of the stack pointer register or of stack memory (FA04), which may force the processor to return to an invalid address from a function (i.e., not to the calling statement).	R, S
FLT-EP-5	Damage of the instruction pointer, instruction pipeline, cache mechanism, code memory, etc. (FA05), which may result in the execution of statements in an invalid order.	R, S
FLT-EP-6	Development Fault - Misunderstood specification (i.e. under-estimation of traffic/load intensity) or classic coding (i.e. bugs in the implementation of the memory management algorithm)	T

*Domain: A = Automotive, R = Railway, S = Space, T = Telecom

2.2.2 Errors

An error is the part of a system's total state that may lead to a failure. An error is detected if its presence is indicated by an error message or error signal. Errors that are present but not detected are latent errors.

The following table identifies the most relevant errors associated with each Asset for the domains targeted by the CHESS project. The errors have been specified in collaboration with the industrial user partners in the project reflecting those that are most common or of most concern.

Errors	Description	Domain*
Processing Unit		
ERR-CPU-1	CPU Services not available	T, S
ERR-CPU-2	CPU time monopolization	T, S
ERR-CPU-3	Blocking of partition due to communication deadlock	A, S
Memory		
ERR-MEM-1	Memory corruption due to unintended writing to memory of other software partition	A,S
Communication		
ERR-COM-2	Loss of peer communication	A, T
ERR-COM-3	Timing errors (too late or too early)	A, S
ERR-COM-6	Message corruption: A transmitted message is altered during transmission over the network (e.g. caused by electromagnetic interferences).	A, R
ERR-COM-7	Masquerading: A subject masquerades itself as another subject to communicate with the EP	A, S
ERR-COM-8	Message loss : A transmitted message is lost during transmission. Can be caused by electromagnetic interferences, buffer overflow, etc	A
ERR-COM-9	Unintended message repetition : The same message is unintentionally send again	A, S
ERR-COM-10	Resequencing : Data arrives in other sequence as it was sent.	A, S
Device		
ERR-DEV-1	Device resources usage overload	T,S
ERR-DEV-2	Device resource provided corrupted services	A
ERR-DEV-3	Device ressources not available	T
Execution Platform code and data		
ERR-EP-1	Execution Platform software services do not work as expected	T,S, A
ERR-EP-2	Invalid or unauthorized configuration parameters	T, S
ERR-EP-3	Data abstraction violation (at compile time)	S
ERR-EP-4	Data constraint violation (at run time)	S
Real Time		
ERR-RT-1	Missed deadline	S
ERR-RT-2	Priority inversion	S

*Domain: A = Automotive, R = Railway, S = Space, T = Telecom

2.2.3 Failures

A failure is a transition from correct service to incorrect service, i.e., to not implementing the system function. The deviation from correct service may assume different forms that are called failure modes. The service failure modes characterize incorrect service according to four viewpoints: the failure domain, the detectability of failures, the consistency of failures, and the consequences of failures on the environment. The domain viewpoint distinguishes between content and timing failures; timing failures may be further distinguished between early or late, depending on whether the service is delivered too early or too late. Failures when both information and timing are incorrect fall into two classes: halt, when the external state becomes constant and erratic otherwise. The detectability viewpoint addresses the signalling of service failures to the user(s). The consistency viewpoint describes the behaviour of the system when it has two or more users. Grading the consequences of the failures upon the system environment enables failure severities to be defined. The failure modes are ordered into severity levels, to which are generally associated maximum acceptable probabilities of occurrence. The number, the labelling, and the definition of the severity levels, as well as the acceptable probabilities of occurrence, are application-related.

The following table identifies the most relevant failures associated with each Asset for the domains targeted by the CHESS project. The failures have been specified in collaboration with the industrial user partners in the project reflecting those that are most common or of most concern.

Failures	Description	Domain*
Processing Unit		
FAIL-CPU-1	Commission failure – unexpected restart of CPU	T
Communication		
FAIL-COM-1	Timing failure (late)-communication is established late w.r.t. what expected	T, A
FAIL-COM-2	Omission failure - No connection establishment	T, A
FAIL-COM-3	Silent failure - communication service is not accessible	T, A
FAIL-COM-4	(Intermittent) Omission failure – Transmission service (degradated) interrupted (Sudden, abnormal, (intermittent) disconnection of already established connections)	T
Execution Platform code and data		
FAIL-EP-1	Omission Failure (Value=null) loss of data (related to **FAIL-COM-4**, as well as, **FLT-COM-1-3**)	T
FAIL-EP-2	Value Failure – Inconsistent Data	T
FAIL-EP-3	Value Failure – data corruption	T, A

*Domain: A = Automotive, R = Railway, S = Space, T = Telecom

3. MITIGATION MEANS OF THE EXECUTION PLATFORM

This sections aims to collect and describe all the mitigation means that are likely to address threats identified in the previous section. It has been elaborated in collaboration with the industrial user partners in the CHESS project reflecting the mitigation means that are most common or of most concern. For each mitigation means, a table is added in order to show which threats are addressed.

The mitigation means are classified into three major categories:

- **Fault prevention**: means to prevent the occurrence or introduction of faults.

- **Fault tolerance**: means to avoid service failures in the presence of faults.
- **Fault removal**: means to reduce the number and severity of faults.
- **Fault forecasting**: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and dependability specifications are adequate and that the system is likely to meet them.

3.1 FAULTS PREVENTION

This section gathers all mechanisms that can be used to avoid threats before they occur. Each identified mechanism addresses one or several threat(s) identified in section 2.2-Threats.

3.1.1 Design principles

- **Separation of privileges :**

The basic principle of privilege separation is to reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the service. This narrows the exposure to bugs in code that is executed with privileges. Ideally, the only consequence of an error in a privilege separated service is denial of service to the adversary himself.

- **Least privilege policy :**

The principle of least privilege is the following: every program and every user should operate using the least amount of privilege necessary to complete the job. Applying the principle to application design limits unintended damage resulting from programming errors. Three approaches are commonly used in application design to help prevent unanticipated consequences from such errors: defensive programming, language enforced protection, and protection mechanisms supported by the execution platform.

- **Fail-safe default policy:**

The "Fail-safe defaults" security policy consists in basing access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965, means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

Threat ID	Threat Description
ERR-CPU-1	CPU Services not available
ERR-CPU-2	CPU time monopolization
ERR-CPU-3	Blocking of partition due to communication deadlock
ERR-MEM-1	Memory corruption due to unintended writing to memory of other software partition
ERR-COM-7	Masquerading: A subject masquerades itself as another subject to communicate with the EP
ERR-DEV-1	Device resources usage overload
ERR-DEV-3	Device ressources not available
ERR-EP-2	Invalid or unauthorized configuration parameters
ERR-EP-3	Data abstraction violation (at compile time)
ERR-EP-4	Data constraint violation (at run time)

3.1.2 Safe programming languages

A large class of faults, errors and failures can be prevented by using "safe programming languages". The crucial characteristic of a safe programming language is that it *protects its abstractions*. Every high-level programming language provides abstractions for accessing machine services. For instance, both Java and Ada provide arrays as one way to access memory. Arrays are associated with operations for accessing and updating them. Safe programming languages ensure that arrays cannot be accessed or updated in other ways, for instance by writing past the memory boundaries of other data structures. Similarly, safe programming languages ensure that records or objects can only be accessed by indexing their fields, or that lexically scoped variables can only be accessed from within their lexical scopes.

Language safety is achieved through *compile-time and run-time checks* that are built into the programming language. Compile-time checks are typically carried out by a *static type-checker*. Safe languages that support dynamic loading of compiled code (e.g., Java) may also support static verification of compiled code at *load-time*. In Java, load-time verification is achieved by a so-called *bytecode verifier*. Roughly speaking, the bytecode verifier ensures that the loaded bytecode could be the result of compiling a well-typed Java program.

Compile-time, load-time and run-time checks detect violations of language abstractions at the point of their occurrence, thus preventing that they turn into latent errors.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments** **Execution platform Language**

		Compile Time	Run Time
ERR-MEM-1	Memory corruption due to unintended writing to memory of other software partition	Yes	Yes
ERR-EP-3	Data abstraction violation (at compile time)	Yes	N/A
ERR-EP-4	Data constraint violation (at run time)	N/A	Yes
FAIL-EP-2	Value failure - inconsistent data	Yes	Yes

3.1.3 Code signing and access control

Access control regulates access to critical system services. It helps in preventing errors by denying services to program components that are not fully trusted because they may be faulty or malicious. Access control helps in *isolating* the effects of untrusted components. Furthermore, it helps in preventing *device overload* (ERR-DEV-1), which may lead to *denial-of-service* (ERR-DEV-3).

Often, access control decisions are partly based on authentication of the subjects that request service. Therefore, *digital code signing* plays an important role in access control, as it constitutes a technique for authenticating program components.

As an example of an access control architecture that is incorporated into a runtime environment, we summarize Java's security architecture [24].

The core component of Java's security architecture is the *security manager*, which mediates access to resources like files, network connections or printers, but also to other critical operations, like shutting down the Virtual Machine (VM), loading native libraries, or replacing the current security manager by another one.

The Java security architecture implements a flexible form of *discretionary access control*. It separates policy expression from policy enforcement. *Policy enforcement* is supported through a method *checkPermission(Permission perm)* that checks if a requested permission is granted. The VM or standard libraries always call this method before executing a security-critical operation. The outcome of this method depends on the current execution context (more on this below).

Policies map program components to permissions. Policy mappings are textually represented in policy files. Policy mappings depend on the following attributes of program components:

1. Who is running the code? (principal)
2. Where does the code come from? (code location)
3. Is the code digitally signed, and if so, by whom? (code signer)

Access decisions are based on *permissions*, rather than exclusion, with a fail-safe default policy. See Section 3.1.1 for a discussion of this design. The default policy is the

empty policy mapping, which does not provide any permissions. Thus, program components that are subject to the default policy execute in a strict *sandbox*.

The outcome of the method *checkPermission(Permission perm)* depends on the current execution context: A permission is granted if and only if all methods that are on the current thread's current call stack have the required permission. In other words, the granted permission is the intersection of all permissions of methods on the current call stack. Thus, checking if a permission is granted requires *stack inspection*. The stack inspection mechanism is an instance of the *least-privilege-principle*, as discussed in Section 3.1.1.

Java's security manager provides a *privileged mode* for escaping from the strict stack inspection mode. The standard example for the need of a privileged mode is a password-checking method. Such a method requires permission to access a password file, but its caller typically does not have this permission. Therefore, an access control mechanism that is strictly based on stack inspection would be too restrictive. To lift this restriction, Java offers a mechanism for executing methods in a privileged mode in order to temporarily escape from the stack inspection mode.

Threat ID	Threat Description
ERR-DEV-1	Device resource usage overload
ERR-DEV-3	Device resources not available

3.1.4 Code analysis method

Static code analysis

Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code.

Dynamic code analysis

Dynamic code analysis is the analysis of computer software that is performed by executing programs built from that software system on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior. Use of software testing techniques such as code coverage helps ensure that an adequate subset of the program's set of possible behaviors has been observed.

Threat ID	Threat Description
FLT-EP-1	Development Fault - Misunderstood specification or classic coding bugs in the implementation of the state-transition machine, whose states correspond to the operation modes and transitions represent the possible transitions between operation modes (FA01). It may result in incorrect initialization or invalid trajectory in the system state space.

Threat ID	Threat Description
FLT-EP-2	Development Fault - Faults in the implementation of the scheduling algorithm (FA02), which may result in incorrect order of task invocations or synchronization issues, as well as load balancing issues.
FLT-EP-6	Development Fault - Misunderstood specification (i.e. under-estimation of traffic/load intensity) or classic coding (i.e. bugs in the implementation of the memory management algorithm)

3.1.5 Load balancing

Load balancing is a technique to distribute workload evenly across two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload.

Threat ID	Threat Description
ERR-CPU-1	CPU Services not available
ERR-CPU-2	CPU time monopolization

3.1.6 Flow control

Flow control is the process of managing the rate of data transmission between two EP to prevent a fast sender from outrunning a slow receiver. It provides a mechanism for the receiver to control the transmission speed, so that the receiving node is not overwhelmed with data from transmitting node.

Threat ID	Threat Description
FLT-EP-6	Development Fault - Misunderstood specification (i.e. under-estimation of traffic/load intensity) or classic coding (i.e. bugs in the implementation of the memory management algorithm)

3.2 FAULTS TOLERANCE

This section gathers mechanisms that can mitigate faults during runtime. Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. It is generally implemented by error detection and subsequent system recovery.

Each identified mechanism addresses one or several threat(s) identified in section 2.2-Threats.

3.2.1 Error detection

Error detection originates an error signal or message within the system. An error that is present but not detected is a latent error.

3.2.1.1 Sanity checks [26]

Sanity checks also known as reasonableness checks, represent an error detection technique. This technique uses known semantic properties of data (e.g. range, rate of change, and sequence) to detect errors. This technique is useful to detect value failures.

Threat ID	Threat Description
ERR-EP-2	Invalid or unauthorized configuration parameters
FAIL-EP-2	Value Failure – Inconsistent Data

3.2.1.2 Reversal checks [26]

Reversal checks represent an error detection technique. This technique uses the output of a module to compute the corresponding inputs based on the function of the module. An error is detected if the computed inputs do not match the actual inputs. This technique is useful to detect value failures.

Threat ID	Threat Description
FAIL-EP-2	Value Failure – Inconsistent Data

3.2.1.3 Control and monitor [28]

Control and monitor is one of the simplest forms of design diversity, which involves using a control component and a monitor component. The monitor component implements a much simpler logic to provide a degraded service compared to the complex service provided by the control component. If the outputs from the two components are equivalent within some tolerance, the results from the control component are deemed correct and used by the system; otherwise the results from the monitor component will be used. Control and monitor is useful to detect value failures.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FAIL-EP-2	Value Failure – Inconsistent Data	
FAIL-EP-3	Value Failure – data corruption	due to **FLT-COM-1**

3.2.1.4 Recovery blocks [29]

The recovery blocks technique consists in providing multiple versions of a software module or component (a primary and one or more alternate modules). These versions are sequentially executed, until either an acceptable output is obtained or the alternates are exhausted. An acceptance test is used for error detection. It is not necessary for all the modules to produce exactly the same results, as long as the results are acceptable as defined by the acceptance test. This technique is useful to detect value failures.

Mitigation Means coverage

Threat ID	Threat Description	Comments
FLT-COM-3	Interaction Fault (External fault) - Transmission path blocked	Usage of another configuration
FAIL-EP-3	Value Failure – data corruption	due to **FLT-COM-1**

3.2.1.5 Watchdog timers (detection only or detection and trigger)[27]

Detection only: Watchdog timer is commonly used to deal with *timing failures*. A timed-constrained process requires to be monitored to reset the timer before it expires as an indication that the process is operating satisfactorily. It can detect whether a task overrun its scheduled processing time, and the absence of outputs from an untrustworthy component.

Detection and trigger: Where the absence of some output is deemed hazardous, a watchdog timer could be used not only to detect the absence of the component output but also to trigger the use of some other logic (e.g. monitor component) to produce an alternative output.

Threat ID	Threat Description
ERR-DEV-1	Device resources usage overload
FAIL-COM-1	Timing failure (late)-communication is established late w.r.t. what expected

3.2.1.6 Replication check

Replication check (also called multiple computation and comparison) is a powerful technique to detect random transient faults, i.e., faults whose presence is bounded in time. The simplest example of replication check is the duplicated computation and comparison, which is capable to detect faults provided they do not have the same effects in the individual computations. In this case comparison is a single point of failure since its fault may render the whole technique useless. Typically, independent (external) comparison or so-called fail-safe comparison techniques are used.

The technique is characterized by the following parameters:

- Coverage, which is typically high as all faults with independent effects in the computations are detected. Note that permanent faults that occurred before the computation may have the same influence on the computations. This phenomenon can be reduced by assuring that the computations are implemented in such a way that the executed code is statically linked and stored in different areas of permanent storage and memory. In this way disk and memory faults are handled, but permanent faults related to the CPU and other hardware devices are not covered.

- Latency, which depends on the length of the computation that is checked. In case of duplicated execution, the average latency is the length of the computation plus the time needed for comparison. However, it is important to note that the technique is capable to assure fail-silent behavior with regard of the covered faults in the checked computation, i.e., it can assure that in case of failure no

service at all is delivered at the service interface (e.g., no messages are sent in a distributed system).

When systematic faults (e.g., software design faults) are also considered then design diversity can be applied. Diversity may include data diversity (e.g., swapping the bytes of the data values, inverting the bits of signal values in the redundant storage etc.) or functional diversity (e.g., implementing different algorithms).

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-*	Random transient faults	Applicable to detect the faults discussed in Section 2.2.1 such that i) their presence is bounded in time, and ii) they have independent effects in the computations

FLT- means all the faults described in the Faults section*

3.2.1.7 Control flow monitoring technique

Control flow monitoring [4] is a technique to detect invalid sequences of program instructions. Compact identifiers (so-called signatures) are assigned to states (branch-free instruction blocks) of the program, and the run-time sequence of these signatures is transferred to a separate hardware or software component, the watchdog processor [5]. This watchdog checks whether the signatures are i) regularly received (implementing a timeout checking that can detect if the program crashed or entered an infinite loop) and ii) they represent a valid control flow (i.e., the state transitions are allowed by a reference control flow that can be derived from the specification or memorized as the set of valid execution traces).

Several types of control flow monitoring techniques can be found in the literature. In modern microprocessors equipped with on-chip pipeline and instruction cache, viable techniques are only those assigning the signatures to program states explicitly, by inserting the signature transfer instructions into the program source. The signature transfer instructions can be inserted into the source of high level programming languages (like C) by an automated pre-processor before compilation.

The reference information that is used to check the run-time sequence of signatures received by the watchdog processor can be based on:

1. The program Control Flow Graph (CFG) extracted from the source code. In this case either the reference CFG is stored in the watchdog processor, or the reference information is embedded in the signatures themselves, i.e., each signature identifies its allowed successors on the basis of the CFG.

2. A high-level specification. In this second case, specification-based checking can be performed by storing the behavioral model (like an UML state diagram or a temporal logic specification) in the watchdog that interprets it and compares the run-time behavior with the specification. This way, implementation errors can be also detected, besides operational errors.

The technique is characterized by the following parameters:

- Coverage, which depends on the frequency of signatures assigned to program states. Experimental analysis showed that up to 70-80% coverage for control flow related errors can be achieved.

- Latency, which depends on the average number of machine instructions between the successive signatures. The typical values are 20..100 instructions in the case of simple C programs.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-EP-1, FLT-EP-2, FLT-EP-3, FLT-EP-4, FLT-EP-5, FLT-EP-6	Faults producing an invalid sequences of program instructions	This MM addresses both software permanent faults introduced during development, and transient or permanent operational hardware faults

3.2.1.8 Cyclic Redundancy Codes

Cyclic Redundancy Codes (CRCs [6]) are widely used in network transmissions and data storage applications because they provide better error detection than lighter weight checksum techniques. Some terms used in the following discussion are:

- Data word - the data that is fed into the CRC computation to produce the checksum.

- Frame Check Sequence (FCS) - the value produced by the CRC computation. It provides the redundant information necessary for error detection.

- Code word - the data word with the FCS appended.

- Undetected error - result of an error which happens to corrupt bits in the code word in such a way that it produces another valid code word. It is important to note that corruptions can and often do occur in both the data word and FCS portions of a code word.

- Burst error - an error pattern stated in terms of a length *m* (i.e., a *m*-bit burst error) where two up to *m* bit errors may occur exclusively in a *m* bit range.

- Hamming Distance (HD) - in the context of error detection, the minimum number of bits in the code word that must be independently corrupted in order to cause an undetected error. For a CRC, the HD depends on the data word length, the FCS length, and the generator polynomial used. For example, the polynomial $x^8+x^5+x^2+x^1+x^0$, which has HD=4 for data words of 18 to 55 bits, will detect all 1-, 2-, and 3-bit errors for those lengths.

While arithmetic checksum codes can only provide HD=2 or HD=3 for most data lengths, CRCs can give much higher HD for the same FCS length. Critical embedded applications usually have a high HD requirement of HD=6 (see [7]).

The FCS is defined by the equation: $crc(x) = (data(x) \bullet x^k) \bmod g(x)$, where g(x) is the "generator polynomial" of order k. A more detailed description can be found in [6].

There are five basic parameters that affect the FCS output by the CRC implementations in real systems: 1) CRC polynomial, 2) initial CRC value, 3) final value which is XORed in the CRC register, 4) order in which data bits are processed, and 5) order in which CRC bits are placed into the FCS field. A partial list of these parameters for various standards can be found in [8], [9]. A change in any of these parameters will affect the final FCS value.

Threat ID	Threat Description
ERR-COM-6	Message corruption: A transmitted message is altered during transmission over the network (e.g. caused by electromagnetic interferences).

3.2.1.9 Diagnostic checks (for RAM, word-oriented memory, flash memory, and CPU)

Diagnostic checks allow the testing of hardware components to identify latent (mainly permanent) faults in the hardware. Such tests are often performed at the start-up of critical systems, and periodically repeated during normal execution. Diagnostic tests are typically characterized by the following parameters:

- Coverage, which depends on the thoroughness of the test procedures with respect of the potential failure modes. The technique is efficient for permanent faults but not for transient faults (transient faults can be detected only if they occur during testing or have long-lasting effects).

- Latency, which depends on the test frequency (periodicity), which is a design parameter and can be used to fine tune the statistical parameters of hazards.

A large set of diagnostic tests is available in the literature, for various hardware components and faults models of a system. In the following, we focus on the fault models and the diagnostic tests for the most important component of a typical system: RAM, word-oriented memory, flash memory, and CPU.

The RAM testing

Amongst the various test algorithms known for RAM (e.g. zero-one, checkerboard, galloping, sliding, butterfly and moving inversion), we focus on march test algorithms [10], [11], which require test times on the order of N (where N is the number of bits on the memory). Algorithms which require test time on order of N^2 or $N^{3/2}$ are impractical for modern high density RAM chips (16 megabits and more).

A **March test algorithm** is a finite sequence of march elements, where a march element is specified by an address order and a number of read/write operations. In the following table, threats covered by widely used marching test algorithms (i.e., MSCAN, MATS,

MATS+, Marching 1/0, MATS++, March X, March C, March C-, Extended March C-) are summarized.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-MEM-1	Stuck-at fault	Stuck-at fault of type OR & AND (using MSCAN test)
FLT-CPU-1	Address-decoder fault	Using MATS test
FLT-MEM-1, FLT-CPU-1	Address-decoder fault and Stuck-at fault	Address-decoder fault and Stuck-at fault of type OR & AND (using MATS+ test)
FLT-MEM-1, FLT-CPU-1, FLT-MEM-2	Address-decoder fault, Stuck-at fault and Transition fault	Using Marching 1/0 test
FLT-MEM-1, FLT-CPU-1, FLT-MEM-2	Address-decoder fault, Stuck-at fault and Transition fault	Using MATS++ test (Goor 1991)
FLT-MEM-1, FLT-CPU-1, FLT-MEM-2, FLT-MEM-5	Address-decoder fault, Stuck-at fault, Transition fault and Coupling fault	Using March X test
FLT-MEM-1, FLT-CPU-1, FLT-MEM-2, FLT-MEM-5	Address-decoder fault, Stuck-at fault, Transition fault and Coupling fault	Using March C test (Marinescu, 1982)
FLT-MEM-1, FLT-CPU-1, FLT-MEM-2, FLT-MEM-5	Address-decoder fault, Stuck-at fault, Transition fault and Coupling fault	Using March C- test (Goor, 1991)
FLT-MEM-1, FLT-CPU-1, FLT-MEM-2, FLT-MEM-5, FLT-MEM-4	Address-decoder fault, Stuck-at fault, Transition fault, Coupling fault and Stuck-open fault	Using Extended March C- test

According to this analysis, Extended March C- test offers the widest fault coverage according to the above mentioned fault model. Note that more refined faults models needs even more complicated march testing algorithms (e.g., in case of so-called linked faults, that influence the behavior of each other, i.e., the behavior of a certain fault can change the behavior of another one such that masking can occur) [11].

The main limitations of March tests are the coverage of sequential faults in address decoders, and the coverage of neighborhood pattern sensitive faults (NPSF). A possible solution is using address sequence variations (hopping or pseudorandom addresses). Note that physical neighborhood shall be known for systematic testing of NPSF, which is not known in modern memory chips, this is why pseudo-random techniques are proposed.

Testing a word-oriented memory

A word-oriented memory has read/write operations that access the memory cell array by a word instead of a bit. Word-oriented memories can be tested by applying a bit-oriented test algorithm repeatedly with a set of different data backgrounds, i.e., background bit is replaced by background word [12]. The repeating procedure multiplies the testing time.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
The same Mitigation Means coverage table as for RAM testing (see previous paragraph)		

Testing a flash memory

Another component that is important to test periodically is the Flash memory. As opposed to the read and the write operations in RAM, the Flash memory has read, program and erase operations. Full testing of Flash memory core is difficult due to i) the customized core and I/O, ii) its isolation (accessibility problems), and iii) the reliability issues: disturbances, over program/erase, under program/erase, data retention, cell endurance, etc. and the relatively long program/erase time.

A RAM test pattern definition includes both the data pattern and the address pattern: the time to read a pattern is the same as the time to write a pattern. For Flash memories, however, the address and data pattern definitions must be segregated: it has long write times relative to the read times. Typical data patterns are solid, checkerboard, random, etc., typical address patterns are address increment/decrement, address complement, column/diagonal galloping, etc.

Testing WPDF (see Section 2.2.1) consists of the following steps (assuming that reading and programming are done column-wise): (1) Flash, (2) program the first column, (3) read all cells except the first column, (4) flash, (5) program any column except the first, (6) read the first column.

Similarly, testing of WEDF (see Section 2.2.1) consists of the following steps: (1) Flash, (2) program all cells, (3) read all cells except the last column, (4) program any column except the last, (5) read the last column.

As example, a March-based Flash test (March-FT for NOR-type Flash memories [23]) covers SAF, TF, SoF, AF, CF (static), WPDF and WEDF (see Section 2.2.1).

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-MEM-1, FLT-MEM-2, FLT-MEM-4, FLT-CPU-1, FLT-MEM-5, FLT-MEM-7, FLT-MEM-8	Stuck-at fault, Transition fault, Stuck-open fault, Address-decoder fault, Coupling fault (static), Word-line program disturbance fault and Word-line erase disturbance fault	Using March-FT test for NOR-type Flash memories

Testing a CPU

The most typical categories of faults for a CPU are ([13]): faults in the instruction decoding and control function, faults in the data transfer function, faults in the data manipulation function, faults in the register decoding function, and faults in the data storage function. A CPU test is typically a program stored in RAM (or ROM), and the challenge is the generation of an effective test program. Some blocks of the CPU can be tested using standard techniques (like cache or multipliers), some others need to be addressed specifically (e.g. a pipeline). There are several approaches in the literature as described in the following:

- The traditional technique of testing microprocessors [13] relies on manual test generation and knowledge of the internal structure of the target processor. Test engineers have to create deterministic test patterns to excite the entire set of operations performed by each internal component of the CPU.

- Later, techniques for semi-automatic synthesis of self-test for stuck-at faults were introduced. For example, the approach in [14] generates a sequence of instructions that enumerates all combinations of the operations and systematically selects operands. The test engineer shall heuristically assign values to instruction operands to achieve high fault coverage. Typically, automatic techniques are able to generate efficient tests for combinational parts (e.g. the ALU), but are less efficient for sequential modules (e.g. pipeline control units). Note that if the exact circuit diagram of the CPU is not available then the tests are restricted to structured functional tests of the CPU modules and data paths.

- Random sequences of instructions are also proposed for testing [15]. This technique is able to attain a fair level of fault coverage. An automatic procedure to create programs that repeatedly generate and execute pseudo-random or directed sequences of machine code, was successfully applied by Intel in case of Pentium 4 and Itanium processors. It also requires experienced test engineers to estimate the number of test sequences to be generated [16].

- Test generation techniques like evolutionary programming (using macros whose parameters are chosen by a genetic algorithm) together with fault simulation are proposed for the purpose of testing non-standard internal components.

A testing technique could either be functional or structural. Functional testing techniques ignore the internal logic of the CPU and generate the test sequence based on the instruction set. Structural testing uses the knowledge of internal hardware for generating the test sequence.

Threat ID	Threat Description
FLT-CPU-2	Faults in the instruction decoding and control function
FLT-CPU-3	Faults in the data transfer function
FLT-CPU-4	Faults in the data manipulation function
FLT-CPU-5	Faults in the register decoding function
FLT-CPU-6	Faults in the data storage function

3.2.1.10 Monitoring of application code execution

Monitoring of application code execution help ensure that the application code conforms to some of its non-functional specifications. It relies on the general architecture of a component based framework, which is illustrated on the following schema:

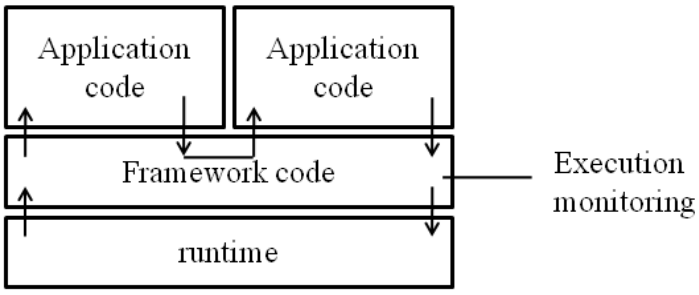


Figure 2 : Component based framework architecture

As all the communications coming to or from the application code go through the framework code, it is possible to set monitors that check the external behaviour of the application code against some specifications.

The framework can implement watchdog timers to monitor the response time of the application code, and either raise an error or send a default response in case of problem. It can also check data validity, either for incoming or outgoing communications.

Mitigation Means coverage	
Threat ID	**Threat Description**
ERR-COM-7	Masquerading: A subject masquerades itself as another subject to communicate with the EP
ERR-COM-8	Message loss : A transmitted message is lost during transmission. Can be caused by electromagnetic interferences, buffer overflow, etc

Mitigation Means coverage	
Threat ID	**Threat Description**
ERR-COM-9	Unintended message repetition : The same message is unintentionally send again
ERR-COM-10	Resequencing : Data arrives in other sequence as it was sent.
ERR-EP-1	Execution Platform software services does not work as expected
ERR-RT-1	Missed deadline
ERR-RT-2	Priority inversion

3.2.2 Errors Handling

Error handling mechanisms are used in association with error detection mechanisms in order to bring a complete and appropriate response to the occurred error.

3.2.2.1 Transaction-based roll-back recovery

To mitigate incompletion failures, atomicity-related approaches are used.
Two-phase commit (2PC), for instance, represents an effective approach to achieve all-or-nothing semantics. 2PC is an atomic commitment protocol. It is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort (roll back) the transaction.
In case the all-or-nothing semantics is considered too strict, a different and ideally customizable approach should be introduced [2].

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
ERR-EP-2	Invalid or unauthorized configuration parameters	configuration data are stored within a Database
FAIL-EP-2	Value Failure – Inconsistent Data	

3.2.2.2 Hardware redundancy

Hardware redundancy consists in the provision of multiple identical instances of the same system and switching to one of the remaining instances in case of a failure (failover).

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-COM-1	Interaction Fault (External fault) - Line transmission disturbance	

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-COM-2	Interaction Fault (External fault) - Signaling link down	
ERR-CPU-1	CPU Services not available	
ERR-COM-6	Message corruption: A transmitted message is altered during transmission over the network (e.g. caused by electromagnetic interferences).	
ERR-COM-8	Message loss : A transmitted message is lost during transmission. Can be caused by electromagnetic interferences, buffer overflow, etc	
ERR-DEV-3	Device resources not available	
FAIL-CPU-1	Commission failure – unexpected restart of CPU	
FAIL-COM-2	Omission failure - No connection establishment	
FAIL-COM-3	Silent failure - communication service is not accessible	
FAIL-COM-4	(Intermittent) Omission failure – Transmission service (degradated) interrupted (Sudden, abnormal, (intermittent) disconnection of already established connections)	
FAIL-EP-1	Omission Failure (Value=null) loss of data (related to **FAIL-COM-4**, as well as, **FLT-COM-1-3**)	
FAIL-EP-3	Value Failure – data corruption	due to **FLT-COM-1**

3.2.2.3 Threshold-based techniques for diagnosis

Heuristic approaches to diagnosis are typically based on simple mechanisms suggested by intuitive reasoning and then validated by experiments; for example, the so called "count–and–threshold" mechanisms count (in some way) error messages collected over time, raising alarms when the counter passes a given threshold. These approaches emerged in those application fields where components oscillate between faulty and correct behavior because a large fraction of faults are transient in nature; in those application fields one-shot diagnosis is not effective, because the cost for treating a transient fault as a permanent one is very disadvantageous.

The most sophisticate representative of the heuristic mechanisms is **α–count** [17], a heuristic proposed for the discrimination between transient and permanent faults. The details about the α–count mechanisms are given hereafter.

The α–count heuristic, proposed for the first time in [18], was conceived for solving the problem of the discrimination between transient, intermittent and permanent faults, taking into account that "components should be kept in the system until the throughput benefit of keeping the faulty component on–line is offset by the greater probability of multiple (hence, catastrophic) faults" (citation from [17]).

The faults and failure modes assumed for the monitored component are the following:

- permanent faults: internal faults which lead the component to fail every time the component is activated.
- intermittent faults: internal faults which show a high occurrence rate and which eventually might turn into permanent faults.
- transient faults: external faults which have an uncorrelated occurrence rate and should not determine the exclusion of the monitored component.

The α–count heuristic implements the "count–and–threshold" criteria, so the rationale behind the α–count is the following: many repeated errors within a little time window are easily seen as evidence for a permanent or an intermittent fault; some isolated errors collected over time could be evidence for transient faults.

α–count hence counts error signals collected over time, raising an alarm when the counter passes a predefined threshold, and decreasing the counter when not-error signals are collected.

Extended studies about the tuning of the internal parameters (K and α_T) are described in [17], where the trade–off between promptness and accuracy is evaluated. Applications and variants (e.g. the double–threshold variant[1]) are described in [19], [20]. Practical applications of the α–count heuristic can be found also in the following works: [21], where α–count has been implemented in the GUARDS[2] architecture for assessing the extent of the damage in the individual channels of a redundant architecture; [22], where α–count is used in a COTS based replicated system to diagnose replica failures based on the result of some voting on the replica output results.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-*	All faults that lead the component to fail every time the component is activated (permanent), or faults which show a high occurrence rate and which eventually might turn into permanent faults (intermittent)	This MM diagnoses if a component is affected by a permanent or intermittent fault

FLT- means all the faults described in the Faults section*

3.2.2.4 Scheduler's protocol PIP and PCP

The Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) are Scheduler's protocol when the scheduling policy is based on the **fixed priority pre-emptive scheduling**. They have been defined to prevent the **priority inversions** and **deadlock on semaphores** in the real-time synchronization of concurrent processes.

Reminder of the priority inversions issue:

[1] The double–threshold mechanism temporarily excludes the monitored component after the first threshold is exceeded, giving an opportunity to be reintegrated; the monitored component is definitely excluded as soon as the second threshold is exceeded.
[2] Generic Upgradable Architecture for Real-time Dependable Systems.

SHA90 [25] :

[Suppose that J1, J2, and J3 are three jobs arranged in descending order of priority with J1 having the highest priority. We assume that jobs J1 and J3 share a data structure guarded by a binary semaphore S. Suppose that at time T1, job J3 locks the semaphore S and executes its critical section. During the execution of job J3's critical section, the high priority job J1 is initiated, preempts J3, and later attempts to use the shared data. However, job J1 will be blocked on the semaphore S. We would expect that J1, being the highest priority job, will be blocked no longer than the time for J3 to complete its critical section. However, the duration of blocking is, in fact, unpredictable. This is because job J3 can be preempted by the intermediate priority job J2…]

Reminder of the deadlock on semaphores:

A deadlock on semaphore occurs when a job J3, with lower priority, after locking a semaphore S1, attempts to lock a semaphore S2, already locked by a job J1, with higher priority and itself blocked on the lock of the semaphore S1

Basic definition of the Priority Inheritance Protocol:

SHA90 [25] :

[The basic idea of Priority Inheritance Protocol is that when a job J blocks one or higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks. After exiting its critical section, job J returns to its original priority level.]

Basic definition of the Priority Ceiling Protocol:

SHA90 [25] :

[The goal of this protocol is to prevent the formation of deadlocks and of chained blocking. The underlying idea of this protocol is to ensure that when a job J preempts the critical section of another job and executes its own critical section z, the priority at which this new critical section z will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections…. This idea is realized by first assigning a priority ceiling to each semaphore, which is equal to the highest priority task that may use this semaphore.]

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
ERR-RT-1	Missed deadline	Prevent missed deadline due to a priority inversion
ERR-RT-2	Priority inversion	Prevent priority inversion as it has been described.
ERR-CPU-3	Blocking of partition due to	Prevent deadlock on semaphore since a

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
	communication deadlock	critical section of a job cannot be preempted by other jobs sharing same semaphores.

3.2.2.5 Timing and space partitioning

Timing and space partitioning as fault prevention software design techniques leads to the development of a system in which each software partition is independently processed from each other. Partitioning a system is achieved along two principal axes:

- Spatial: processor (in terms of node, physical or virtual), memory;
- Temporal: processor cycles when processor time is shared.

Among other benefits, partitioning allows the fault containment in order to prevent fault propagation and to secure information. The partitioning may also help to safely share other peripherals resources without conflict.

Different kinds of hardware architectures must be considered:

- Single-node:
  - Mono-processor: processor time, memory and resources are shared;
  - Multi-core: memory and resources are shared;
- Multi-node: distributed system without shared resource.

In every case, spatial partitioning consists of allocating a dedicated resource to a specific software active component. On a mono-processor, a software active component in one resource partition can't write into the memory of another partition (active or inactive).

Temporal partitioning consists of allocating a dedicated slice of time to a specific software active component when the processor time is shared. The active component may itself be constituted of active sub-components. In such case and inside a partition the scheduling of each active sub-component depends on the partition runtime itself. The activities in one partition do not affect the timing of other partition.

Such partitioning may be achieved at different levels:

- Hardware: For example, the Memory Management Unit (MMU) which ensures the memory partitioning. Such mechanism prevents unauthorized memory access across the partition;

- Software: Dedicated kernel which ensures the timing and/or space partitioning segregation. A kernel providing fully "timing and space" partitioning is an environment similar to a distributed system.

The following figures illustrate different kinds and levels of timing and space partitioning depending on the subjacent hardware:

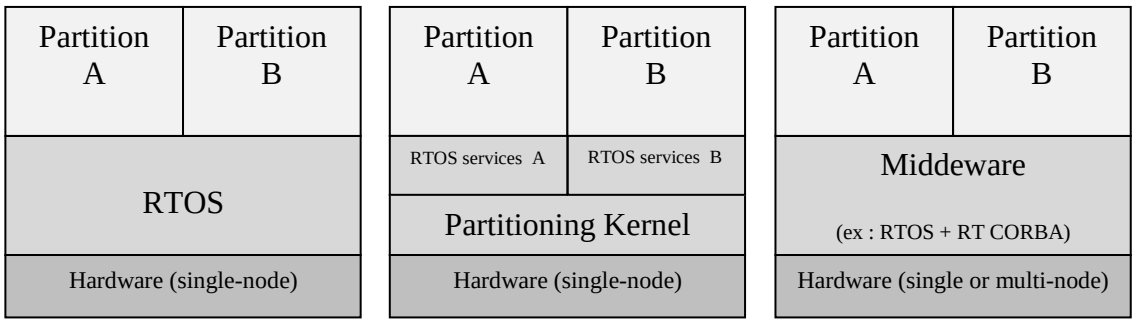


Figure 3 : Partitioning techniques

From the left to the right:

The first schema illustrates partitions built on a conventional Real-Time Operating System (RTOS). RTOS may take advantage of a MMU mechanism of the hardware to enforce a certain level of space partitioning on a single node system. Temporal partitioning may be enforced by the RTOS functionalities. RTOS functionalities may be enforced by the use of a programming language such as Ada 2005, for which the RTOS is subjacent. In such case, the language may provide time and space partitioning at the programming level.

The second schema illustrates another alternative and aims to enforce the temporal and spatial partitioning. Such an alternative is the well known approach of ARINC 653 (pioneer approach) or the relatively recent approach "Multiple Independent Levels of Security" (MILS). Even if both approaches differ by their objective, MILS by addressing isolation for security and safety purposes and ARINC by addressing specifically isolation for safety purposes, both approaches are based on the same principles:

- Space partitioning: each partition is bound to a distinct memory component;
- Timing partitioning: guarantee that each partition is periodically scheduled;
- Inter-partition communication: communication link is allocated to each partition.

The third schema illustrates the ideal system from the point of view of partitioning with ideal separation and isolation means. It requires the use of real-time middleware including a communication link such as, for example, RT CORBA technology. It must be noted that such architecture may also be supported by a MILS approach.

As one of the mitigation means at system level, the "timing and space partitioning" strategy may clearly provide at most error containment inside a partition. Depending on the architecture and the error raised, the containment can be achieved or not as described in the following table:

Mitigation Means coverage				
Threat ID	**Threat Description**	**Comments** *Can the error be contained?*		
		RTOS	**MILS**	
			Single-node	**Multi-node**
ERR-CPU-1	CPU Services not available	Can't	Can't	Could
ERR-RT-1	Missed deadline	Should	Shall	Shall
ERR-MEM-1	Memory corruption due to unintended writing to memory of other software partition	Should	Shall	Shall
ERR-CPU-2	CPU time monopolization	Should	Shall	Shall
ERR-DEV-1	Device resources usage overload	Should	Shall	Shall

3.2.2.6 System reconfiguration

System reconfiguration occurs when the system is no longer capable of delivering the same service as before, i.e. when a fault leads to a **component failure**. But the system may be seen as a set of components and the reconfiguration strategy may depend on the level at which the service failure occurs.

Here after the steps leading to a reconfiguration in a fault treatment:

Fault treatment		
Step 1	**Step 2**	**Step 3**
Fault diagnosis	Error passivation (isolation)	**Reconfiguration**
Determines the cause of error	Prevents the component(s) identified as being faulty from being invoked in further executions	If the system is no longer capable of delivering the same service as before, then a reconfiguration may take place

Depending on the level at which the **component failure** occurs and the subjacent **execution platform mechanisms**, the **reconfiguration** may be achieved by (mainly but not exhaustively):

- Using the faulty component in degraded mode and ensuring its unavailable services are provided by others components;

- Isolating the faulty component and continue the global mission with the remaining components.
- Re-initialization of the application node as a whole, the component, or sub-component;
- Replacing the faulty component by a non-faulty one (at component level or node level);

The reconfiguration strategy depends also on the acceptable latency and the level of the related component and services recovery. A reconfiguration may be processed on a component of the execution platform (e.g. communication link, I/O) with a minimal impact on the application. But a reconfiguration may not be processed on an active component of the application (e.g. task could be restarted at execution platform level).

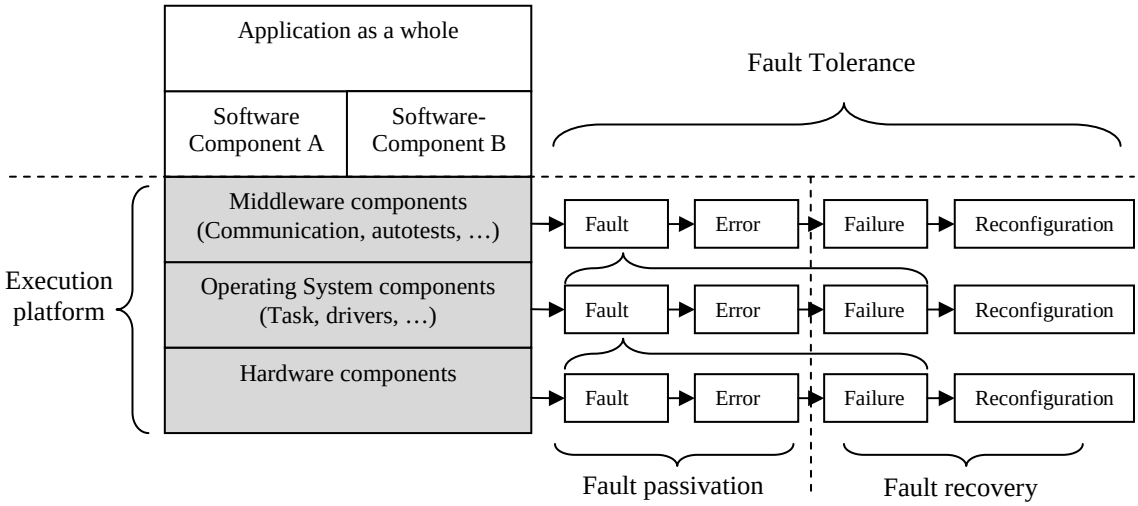


Figure 4 : Reconfiguration strategy

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
ERR-EP-1	Execution Platform software services do not work as expected	Reconfiguration strategy depends on the level at which the service failure occurs

3.3 FAULTS REMOVAL

This section gathers all mechanisms that can help to remove detected faults during runtime. Each identified mechanism addresses one or several threat(s) identified in section 2.2- Threats.

3.3.1 Preventive and corrective maintenance

Software patches functionality is foreseen early during a software development.

This functionality can be used to erase dormant faults identified after the commissioning of the system (preventive maintenance), or to remove active faults detected during runtime (corrective maintenance).

Specific mechanisms are developed and embedded into the final target application or platform to manage the software patches:

- Patch TC acceptance (with checksums)
- Patch storage and activation :
    - Store in RAM (Random Access Memory) before applying
    - Store in SGM (Safeguard Memory) or Mass-memory for persistence across reboots and across processors
    - Store in EEPROM (Electrically Erasable Programmable Read-Only Memory) for permanent storage
- Management of a status table (loaded patch, activation flags, …)
- Free tasks (called free functions) are allocated in the "on board software" and used by software patches requiring to perform unforeseen cyclic activities.

Software patch is, in some cases, the only feasible corrective maintenance mechanisms. For instance, one can not perform ground maintenance on space systems. Patches are used to recover from an unexpected failure or error:

- Correction of a system specification bug: Bad or missing requirements
- Correction of a software bug: Unforeseen branch, Bad data protection…

Patches are also used to adapt the software for a new configuration:

- Change in the equipment due to environment conditions or aging: modification of threshold, timer …
- Change in the equipment due to failures: Elaborate a new control algorithm based on a limited set of sensors/actuators
- Unexpected configuration due to launch failure:

Finally, patches can be used to download unforeseen data to perform investigation.

Patches are realized by creating a new binary image and by uploading the binary differences between the current image and the new one. The new image is elaborated:

- Manually and directly in assembly language when it is simple: change in a parameter (threshold, timer…), suppression of an operation call, a condition…
- By modifying the source code and recompilation if it is more complex: Replace a complete operation, addition of a component

A special environment is available to minimize the differences between the old image and the new one:

- Coding rules
- Compiler and linker directives
- Automatic diff tools
- Granularity is the function

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-EP-1	Development Fault - Misunderstood specification or classic coding bugs in the implementation of the state-transition machine, whose states correspond to the operation modes and transitions represent the possible transitions between operation modes (FA01). It may result in incorrect initialization or invalid trajectory in the system state space.	
FLT-EP-2	Development Fault - Faults in the implementation of the scheduling algorithm (FA02), which may result in incorrect order of task invocations or synchronization issues, as well as load balancing issues.	
FLT-EP-3	Damage of registers or memory storing synchronization variables (FA03), which may result in incorrect order of task invocations or synchronization issues.	
FLT-EP-4	Damage of the stack pointer register or of stack memory (FA04), which may force the processor to return to an invalid address from a function (i.e., not to the calling statement).	
FLT-EP-5	Damage of the instruction pointer, instruction pipeline, cache mechanism, code memory, etc. (FA05), which may result in the execution of statements in an invalid order.	
FLT-EP-6	Development Fault - Misunderstood specification (i.e. under-estimation of traffic/load intensity) or classic coding (i.e. bugs in the implementation of the memory management algorithm)	
ERR-CPU-2	CPU time monopolization	
ERR-CPU-3	Blocking of partition due to communication deadlock	

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
ERR-MEM-1	Memory corruption due to unintended writing to memory of other software partition	
ERR-DEV-1	Device resources usage overload	
ERR-EP-1	Execution Platform software services do not work as expected	
ERR-EP-2	Invalid or unauthorized configuration parameters	
ERR-EP-4	Data constraint violation (at run time)	
ERR-RT-1	Missed deadline	
ERR-RT-2	Priority inversion	

3.4 FAULT FORECASTING

This section gathers all mechanisms that allow estimating the present number, the future incidence, and the likely consequences of faults. Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. This evaluation is quantitative or probabilistic, which aims to evaluate in terms of probabilities the extent to which some of the attributes of dependability are satisfied; those attributes are then viewed as measures of dependability.

Each identified mechanism addresses one or several threat(s) identified in section 2.2-Threats.

3.4.1 Dynamic Dependability Analysis

Dynamic dependability analysis is the analysis of failure modes of a corresponding initial error during scenario execution on a virtual or a real execution platform. In the following, only the virtual execution platform for usage in early development phases is considered.

In the first step of the analysis the execution model is configured as described by a scenario definition and started. In the second step errors are injected to the simulation. The results of the scenario are monitored and checked during execution. If predefined assertions are mismatched a failure of the considered system has occurred. By analyzing the traces and or a step by step execution it will be possible to classify these failures. The used mechanisms and components are explained in more detail in the following sections.

Error Injection

For the simulation of different errors scenarios and the determination of the corresponding failure mode errors must be injected to the simulation. Therefore different elements of the EP must be expanded with feasible error injectors. These error injectors must be fully configurable to introduce different kind of errors to the

simulation at a predefined time. The methods for error injection are described in the following:

- **Mutant**: An error free module is exchanged by an erroneous module to simulated permanent errors.
- **Saboteur**: An error free module is controlled by an additional module to inject permanent or transient errors e.g. bit flips on a transmission channel at a predefined time.
- **Simulator Commands**: Allows variable manipulation during simulation or code execution similar to a debugger.

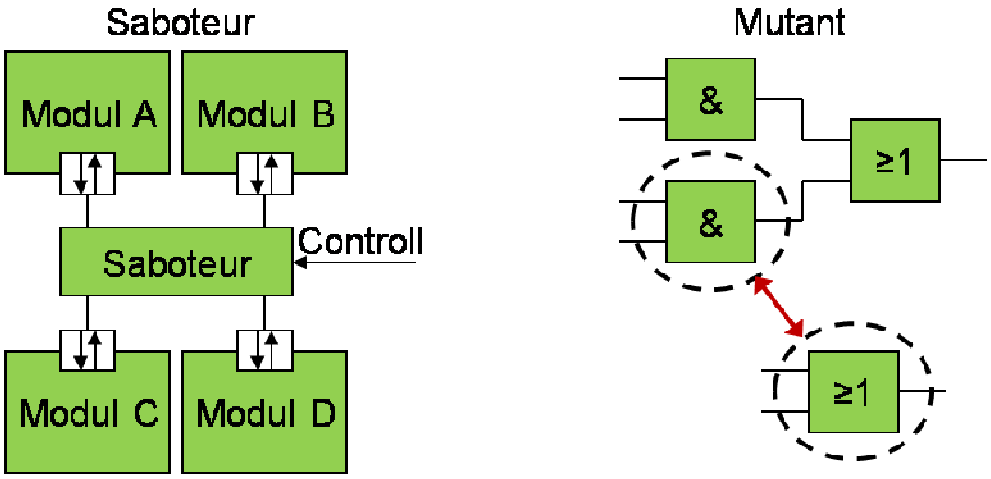


Figure 5 : Error injection methods

Monitoring, Tracing and Result Checking
During simulation it is necessary to monitor different important elements of the system. The data processed by the monitors must be stored in trace files for later analysis. During and after the simulation predefined assertions such as the scenario results must be checked. If a mismatch occurs a failure is located. The classification can be derived by considering the initial errors and the trace data.

Scenario Configuration and Execution
For determining failure probabilities from the execution platform, millions of different simulations must be executed. To accelerate the execution, the simulation must be configurable dynamically, this means without additional compilation or modifications to the simulated hardware or software. Configuration must not include the scenario configuration but at least the configurations of the error injectors.

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
FLT-COM-1	Interaction Fault (External fault) - Line transmission disturbance	
FLT-COM-2	Interaction Fault (External fault) - Signaling link down	
FLT-COM-3	Interaction Fault (External fault) -	

Mitigation Means coverage		
Threat ID	**Threat Description**	**Comments**
	Transmission path blocked	
ERR-COM-2	Loss of peer communication	
ERR-COM-3	Timing errors (too late or too early)	
ERR-COM-6	Message corruption:. A transmitted message is altered during transmission over the network (e.g. caused by electromagnetic interferences).	
ERR-COM-7	Masquerading: A subject masquerades itself as another subject to communicate with the EP	
ERR-COM-8	Message loss : A transmitted message is lost during transmission. Can be caused by electromagnetic interferences, buffer overflow, etc	
FAIL-COM-1	Timing failure (late)-communication is established late w.r.t. what is expected	
FAIL-COM-2	Omission failure - No connection establishment	
FAIL-COM-3	Silent failure - communication service is not accessible	
FAIL-EP-3	Value Failure – data corruption	Either the transmitted user data or the control data of a communication message is corrupted

4. MITIGATIONS MEANS COVERAGE

This section aims to summarize the identified mitigation means in the previous section.

The table below aims to check that each mitigation mean listed in section 3- Mitigation Means of the execution platform addresses (mitigated or prevented) at least one of the threats identified.

Mitigation Means coverage	
Faults prevention	***Threats***
Design principles	ERR-CPU-1, ERR-CPU-2, ERR-CPU-3, ERR-MEM-1, ERR-COM-7, ERR-DEV-1, ERR-DEV-3, ERR-EP-2, ERR-EP-3, ERR-EP-4
Safe programming languages	ERR-MEM-1, ERR-EP-3, ERR-EP-4, FAIL-EP-2
Code signing and access control	ERR-DEV-1, ERR-DEV-3
Static analysis method	FLT-EP-1, FLT-EP-2, FLT-EP-6
Load balancing	ERR-CPU-1, ERR-CPU-2
Flow control	FLT-EP-6
Faults tolerance	***Threats***
Sanity checks	ERR-EP-2, FAIL-EP-2
Reversal checks	FAIL-EP-2
Control and monitor	FAIL-EP-2, FAIL-EP-3
Recovery blocks	FLT-COM-3, FAIL-EP-3
Watchdog timers (detection only or detection and trigger)	ERR-DEV-1, FAIL-COM-1
Replication check	FLT-*
Control flow monitoring technique	FLT-EP-1, FLT-EP-2, FLT-EP-3, FLT-EP-4, FLT-EP-5, FLT-EP-6
Cyclic Redundancy Codes	ERR-COM-6
Diagnostic checks (for RAM, word-oriented memory, flash memory, and CPU) (UNIFI)	FLT-MEM-1, FLT-CPU-1, FLT-MEM-2, FLT-MEM-5, FLT-MEM-4, FLT-MEM-7, FLT-MEM-8, FLT-CPU-2, FLT-CPU-3, FLT-CPU-4, FLT-CPU-5, FLT-CPU-6
Monitoring of application code execution	ERR-COM-7, ERR-COM-8, ERR-COM-9, ERR-COM-10, ERR-EP-1, ERR-RT-1, ERR-RT-2
Transaction-based roll-back recovery	ERR-EP-2, FAIL-EP-2
Hardware redundancy	FLT-COM-1, FLT-COM-2, ERR-CPU-1, ERR-COM-6, ERR-COM-8, ERR-DEV-3, FAIL-CPU-1, FAIL-COM-2, FAIL-COM-3, FAIL-COM-4, FAIL-EP-1, FAIL-EP-3
Threshold-based techniques for diagnosis	FLT-*

Scheduler's protocol PIP and PCP	ERR-RT-1, ERR-RT-2
Timing and space partitioning	ERR-CPU-1, ERR-RT-1, ERR-MEM-1, ERR-CPU-2, ERR-DEV-1
System reconfiguration	ERR-EP-1
Faults removal	***Threats***
Preventive and corrective maintenance	FLT-EP-1, FLT-EP-2, FLT-EP-3, FLT-EP-4, FLT-EP-5, FLT-EP-6, ERR-CPU-2, ERR-CPU-3, ERR-MEM-1, ERR-DEV-1, ERR-EP-1, ERR-EP-2, ERR-EP-4, ERR-RT-1, ERR-RT-2
Faults forecasting	***Threats***
Dynamic Dependability Analysis	FLT-COM-1, FLT-COM-2, FLT-COM-3, ERR-COM-2, ERR-COM-3, ERR-COM-6, ERR-COM-7, ERR-COM-8, FAIL-COM-1, FAIL-COM-2, FAIL-COM-3, FAIL-EP-3

The table below aims to check that each threat (fault, error and failure) listed in section 2.2- Threats is addressed (mitigated or prevented) by at least one of the mitigation means identified.

Threats coverage	
Faults	***Mitigations Means***
FLT-CPU-1	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-CPU-2	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-CPU-3	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-CPU-4	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-CPU-5	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-CPU-6	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-MEM-1	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-MEM-2	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-MEM-3	Replication check, Threshold-based techniques for diagnosis
FLT-MEM-4	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-MEM-5	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-MEM-6	Replication check, Threshold-based techniques for diagnosis
FLT-MEM-7	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-MEM-8	Replication check, Diagnostic checks, Threshold-based techniques for diagnosis
FLT-COM-1	Replication check, Hardware redundancy, Threshold-based techniques for diagnosis, Dynamic Dependability Analysis
FLT-COM-2	Replication check, Hardware redundancy, Threshold-based techniques for diagnosis, Dynamic Dependability Analysis
FLT-COM-3	Recovery blocks, Replication check, Threshold-based techniques for diagnosis, Dynamic Dependability Analysis
FLT-DEV-1	Replication check, Threshold-based techniques for diagnosis
FLT-EP-1	Replication check, Control flow monitoring technique, Threshold-based techniques for diagnosis, Preventive and corrective maintenance , Static analysis method

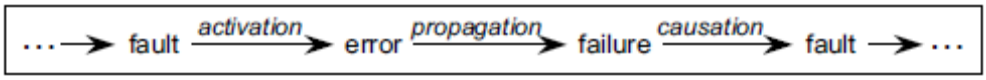
FLT-EP-2	Replication check, Control flow monitoring technique, Threshold-based techniques for diagnosis, Preventive and corrective maintenance, Static analysis method
FLT-EP-3	Replication check, Control flow monitoring technique, Threshold-based techniques for diagnosis, Preventive and corrective maintenance
FLT-EP-4	Replication check, Control flow monitoring technique, Threshold-based techniques for diagnosis, Preventive and corrective maintenance
FLT-EP-5	Replication check, Control flow monitoring technique, Threshold-based techniques for diagnosis, Preventive and corrective maintenance
FLT-EP-6	Replication check, Control flow monitoring technique, Threshold-based techniques for diagnosis, Preventive and corrective maintenance, Static analysis method , Flow Control
Errors	***Mitigations Means***
ERR-CPU-1	Design principles, Hardware redundancy, Timing and space partitioning, Load balancing
ERR-CPU-2	Design principles, Timing and space partitioning, Preventive and corrective maintenance, Load balancing
ERR-CPU-3	Design principles, Preventive and corrective maintenance
ERR-MEM-1	Design principles, Safe programming languages, Timing and space partitioning, Preventive and corrective maintenance
ERR-COM-2	Dynamic Dependability Analysis
ERR-COM-3	Dynamic Dependability Analysis
ERR-COM-6	Cyclic Redundancy Codes, Hardware redundancy, Dynamic Dependability Analysis
ERR-COM-7	Design principles, Monitoring of application code execution, Dynamic Dependability Analysis
ERR-COM-8	Hardware redundancy, Monitoring of application code execution, Dynamic Dependability Analysis
ERR-COM-9	Monitoring of application code execution
ERR-COM-10	Monitoring of application code execution
ERR-DEV-1	Design principles, Code signing and access control, Watchdog timers, Timing and space partitioning, Preventive and corrective maintenance
ERR-DEV-2	
ERR-DEV-3	Design principles, Code signing and access control, Hardware redundancy
ERR-EP-1	System reconfiguration, Monitoring of application code execution, Preventive and corrective maintenance
ERR-EP-2	Design principles, Sanity checks, Transaction-based roll-back recovery, Preventive and corrective maintenance
ERR-EP-3	Design principles, Safe programming languages
ERR-EP-4	Design principles, Safe programming languages, Preventive and corrective maintenance
ERR-RT-1	Timing and space partitioning, Monitoring of application code execution, Preventive and corrective maintenance, Scheduler's protocol PIP and PCP
ERR-RT-2	Monitoring of application code execution, Preventive and corrective maintenance, Scheduler's protocol PIP and PCP
Failures	***Mitigations Means***
FAIL-CPU-1	Hardware redundancy
FAIL-COM-1	Watchdog timers, Dynamic Dependability Analysis
FAIL-COM-2	Hardware redundancy, Dynamic Dependability Analysis
FAIL-COM-3	Hardware redundancy, Dynamic Dependability Analysis
FAIL-COM-4	Hardware redundancy
FAIL-EP-1	Hardware redundancy
FAIL-EP-2	Safe programming languages, Sanity checks, Reversal checks, Control and monitor, Transaction-based roll-back recovery

FAIL-EP-3	Control and monitor, Recovery blocks, Hardware redundancy

5. CONCLUSION

Over the course of the past fifty years many means to attain the dependability of embedded systems have been developed. The list of mitigation means identified in this document aims to address the CHESS specific domains. The approach has been to identify and describe the mitigation means that are likely to be encountered by Execution Platforms for four targeted domains in the CHESS project: Automotive, Space, Telecom and Railway.

These threats are classified into three categories (faults, errors and failures) depending on their situation in the following propagation chain:



The fundamental chain of dependability threats

The mitigation means are also grouped into three major categories:

- **Fault prevention**: means to prevent the occurrence or introduction of faults.
- **Fault tolerance**: means to avoid service failures in the presence of faults.
- **Fault removal**: means to reduce the number and severity of faults.
- **Fault forecasting**: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and dependability specifications are adequate and that the system is likely to meet them.

Thus, this document should help Execution Platform providers of the CHESS consortium to select, in a relevant way, runtime mechanisms that correspond to the project needs in order to attain the appropriate dependability level. It should also help to make these choices by presenting the threats addressed by each identified mitigations means.

Other aspects should be taken in account when one selects mitigation means:

- Cost: the development and enforcement of the solution could be very costly in terms of development time and/or human resources.
- Predictability: some mechanisms could allow attaining a certain dependability level to the detriment of the performances or the predictability of the system.

That is why one must make compromises between the desired dependability level and the feasibility to reach it when these other aspects are also considered.

6. REFERENCES

[1] Ye, F.; Kelly, T.; , "Component failure mitigation according to failure type," *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International* , vol., no., pp.258-264 vol.1, 28-30 Sept. 2004 (MDH)

[2] Wijnand Derks, Willem Jonker, Juliane Dehnert, Paul Grefen, "Customized Atomicity Specification for Transactional Workflows," codas, pp.140, Third International Symposium on Cooperative Database Systems for Advanced Applications (codas), 2001 (MDH)

[3] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, Volume: 1, Issue: 1, Page(s): 11–33, 2004.

[4] Ravishankar K. Iyer and Zbigniew Kalbarczyk. Hardware and Software Error Detection. University of Illinois.

[5] A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors – a survey. Computers, IEEE Transactions on, 37(2):160–174, Feb. 1988.

[6] T. V. Ramabadran and S. S. Gaitonde. A tutorial on CRC computations. IEEE Micro, 8(4):62–75, 1988.

[7] Ray, J. and Koopman, P. 2006. Efficient High Hamming Distance CRCs for Embedded Networks. In Proceedings of the international Conference on Dependable Systems and Networks (June 25 - 28, 2006). DSN. IEEE Computer Society, Washington, DC, 3-12.

[8] M. Barr. Slow and steady never lost the race. Embedded Systems Programming, pages 37–46, January 2000.

[9] R. Williams. A painless guide to CRC error detection. Online: http://www.ross.net/crc/download/crc v3.txt, 1993.

[10] A.J. van de Goor: Testing Semiconductor Memories, Theory and Practice, John Wiley & Sons, Chichester, UK, 1991.

[11] A.J. van de Goor, G. Gaydadjiev: March LR: A Memory Test for Realistic Linked Faults. In Proc. IEEE VLSI Test Symposium, pp. 272-280, 1996.

[12] A. J. van de Goor and I. B. S. Tlili. March tests for word-oriented memories. In DATE ’98: Proceedings of the conference on Design, automation and test in Europe, pages 501–509, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Thatte, S. M. and Abraham, J. A. 1980. Test Generation for Microprocessors. IEEE Trans. Comput. 29, 6 (Jun. 1980), 429-441.

[14] Jian Shen and Jacob A. Abraham. Native mode functional test generation for processors with applications to self test and design validation. In ITC ’98:Proceedings of the 1998 IEEE International Test Conference, pages 990–999, Washington, DC, USA, 1998. IEEE Computer

Society.

[15] Ken Batcher and Christos Papachristou. Instruction randomization self test for processor cores. In VTS ’99: Proceedings of the 1999 17TH IEEE VLSI Test Symposium, Washington, DC, USA, 1999. IEEE Computer Society.

[16] P. Parvathala, K. Maneparambil, W. Lindsay: FRITS – a Microprocessor Functional BIST Method. In Proc. International Test Conference, pp. 590-598, 2002.

[17] Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico & Fabrizio Grandoni. Threshold-Based Mechanisms to Discriminate Transient from Intermittent Faults. IEEE Transactions on Computers, vol. 49, no. 3, pages 230–245, 2000.

[18] Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico & Fabrizio Grandoni. Discriminating Fault Rate and Persistency to Improve Fault Treatment. In 27th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-27), pages 354–362, Seattle, Washington, USA, June 25-27 1997.

[19] Andrea Bondavalli, Silvano Chiaradonna, Domenico Cotroneo & Luigi Romano. Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications. IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 4, pages 223–237, 2004.

[20] Marco Serafini, Andrea Bondavalli & Neeraj Suri. Online Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters. IEEE Transactions on Dependable and Secure Computing, vol. 4, no. 4, pages 295–312, 2007.

[21] David Powell, Christophe Rabéjac & Andrea Bondavalli. Alpha-count mechanism and inter-channel diagnosis. Technical report, ESPRIT Project 20716 GUARDS Report, N°I1SA1.TN.5009.E, 1998.

[22] Luigi Romano, Andrea Bondavalli, Silvano Chiaradonna & Domenico Cotroneo. Implementation of Threshold-based Diagnostic Mechanisms for COTS-based Applications. In 21st IEEE Symposium on Reliable Distributed Systems (SRDS’02), pages 296–303, Osaka University, Suita, Japan, October 13-16 2002.

[23] Jen-Chieh Yeh, Chi-Feng Wu, Kuo-Liang Cheng, Yung-Fa Chou, Chih-Tsun Huang, and Cheng-WenWu. Flash memory built-in self-test using march-like algorithms. In DELTA ’02: Proceedings of the The First IEEE International Workshop on Electronic Design, Test and Applications (DELTA ’02), page 137, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Li Gong, Gary Ellison, Mary Dageforde. Inside Java 2 Platform Security: Architecture, API Design and Implementation. Addison-Wesley, 2nd Edition, 2003.

[25] LUI SHA, RAGUNATHAN RAJKUMAR, JOHN P.LEHOCZKY: Priority Inheritance Protocols: An Approach to Real-Time Synchronisation, IEEE Transactions on computers, Vol. 39, NO. 9, September 1990.

[26] P. A. Lee and T. Anderson. Fault Tolerance: Principles and Practice, 2nd ed.: Springer-Verlag/Wien, 1990.

[27] W. Torres-Pomales. Software Fault Tolerance: A Tutorial," Langley research Center, Hampton, Virginia, Technical Report NASA/TM-2000-210616, 2000.

[28] B. P. Douglass. Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns. Addison Wesley, 1999.

[29] James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. In Operating Systems, Proceedings of an International Symposium, Erol Gelenbe and Claude Kaiser (Eds.). Springer-Verlag, London, UK, 171-187. 1974.