NOTE TO USERS

Addressing Fault Tolerance in Software Development: A Comparative Study

Sadaf Mustafiz

School of Computer Science
McGill University, Montréal

June 2004

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science.

Your file Votre référence
ISBN: 0-494-06428-5
Our file Notre référence
ISBN: 0-494-06428-5

Abstract

Current mainstream software engineering methods do not consider fault tolerance in the requirements engineering and analysis stage. If at all, they only address it much later in the development cycle. However, most modern systems can benefit from some form of fault-tolerance. Especially, complex, concurrent, distributed, or heterogeneous applications are likely to contain software design faults that can lead to system failures. In case of real-time or safety-critical systems, such faults can also result in catastrophes.

This thesis aims to investigate whether software development approaches have integrated the concern of fault tolerance into the early software development stages to satisfy dependability requirements. Software development methods, frameworks, middleware, and other proposed approaches have been studied and are discussed with particular focus on methodological support. Not surprisingly, most approaches are specialized, targeting distributed, real-time, and embedded systems. Finally, a comparison of the various approaches, based on several criteria, is presented.

Résumé

Les processus de développement de logiciels courants ne s'occupent pas de la tolérance aux pannes. Tel est le cas malgré le fait que les logiciels modernes, tels que les systèmes distribués, concurrents ou hétérogènes, sont compliqués à développer, et il est ainsi d'autant plus probable qu'ils contiennent des fautes de conception qui pourraient mener à des défaillances importantes.

Ce travail de maîtrise regroupe des approches spécialisées publiées de nos jours intégrant, d'une manière ou d'une autre, la tolérance aux pannes dans le processus de développement d'un logiciel. Les approches étudiées sont des méthodes de développement, frameworks, middleware, et d'autres techniques qui tentent d'augmenter la fiabilité d'un logiciel pendant son développement. Tout au long de l'étude, le support méthodologique offert est mis en valeur. Non sans surprise, la majorité des approches vise le développement de systèmes embarqués, temps-réels, et distribués. Pour conclure, une comparaison des différentes approches a été établie.

Acknowledgments

I would sincerely like to thank my supervisor, Prof. Jörg Kienzle, for his guidance, assistance, and his very valuable feedback during the writing of this thesis. I am grateful to him for the financial support he provided during the course of this work. I thank him for giving me the opportunity to work with him and for teaching me good research practices.

I would like to thank Dr. Dondossola Giovanna, Dr. Geert Deconinck, and Dr. Alan Burns for providing me with to some very useful reports, which were not publicly available.

I would like to thank my friends for their encouragement and suggestions.

I wish to express my deepest thanks to my family, for their continuous support, encouragement, and prayers.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

"

Your satellite is finally flying. You've waited 10 years for this. The mission is on track and all systems are "go". The REE[1] number cruncher, high above, sifts through and analyzes the torrent of data streaming into your system, giving you on-the-spot data reduction for downlink. You sit in your ground station, delighted as you watch the picture develop.

Out of the blue comes a galactic cosmic ray, hurtling towards you with 500MeV of silicon-pulverizing energy. It hits processor A5's stack pointer. A bit gets flipped. The processor is going to crash and take 25% of your data with it. A second cosmic ray glances off the satellite structure and showers your memory with upset inducing radiation. Your data is being cooked.

What will you do ... what will you do????

" [REE98].

This is where fault tolerance comes in – not just for mission-critical or safety-critical systems, but for any dependable system.

1.1 The Need for Fault Tolerance

Modern applications must respond to an increasing number of requirements. To satisfy user expectations, applications offer more and more functionality, and hence grow more complex. Elaborate user interfaces, multi-media features or interaction with real-time devices, e.g. sensors, require software to promptly and reliably respond to external

[1] NASA **R**emote **E**xploration and **E**xperimentation Project

stimuli and to be able to perform several operations simultaneously. The popularity of the Internet and the growing field of e-commerce have led to an explosion of the number of distributed systems in operation, an increasing number of which are heterogeneous.

Applications like the ones mentioned above must usually provide highly available services. Unfortunately, complex applications are more likely to contain so-called software design faults that are not detected during system testing, and that might at some point lead to system failure. In distributed systems, the probability that a node-failure occurs increases with the number of nodes in the system. Network congestion and partitions are very common in distributed systems, just as are overloaded resources, e.g. devices or databases. Dynamic systems, i.e. applications that handle a potentially unlimited number of clients, have to deal with irregular load. Heterogeneous systems often register abnormal behavior of individual subsystems or components, etc. In short, most modern systems must provide or can at least benefit from some form of fault-tolerance. Surprisingly enough, fault tolerance is not addressed by current mainstream software engineering methods. Software development approaches are discussed in the next section. In general, fault tolerance is considered a "non-functional" requirement, and therefore introduced too late during the development of an application. Most of the time, developers start thinking about fault tolerance only after the main part of the application has already been implemented. In such a situation it is sometimes very hard or even impossible to provide acceptable fault tolerance. Ad-hoc solutions result in complex system structure, hard-to-maintain code and poor performance.

1.2 Software Development Approaches

Software development can be approached in several ways: software development methods, middleware approach, frameworks and middleware framework approach. At times, software development concentrates on providing domain-specific software architectures.

Software development methods define a step-by-step process that leads application developers from the elaboration of an initial requirements document through to the final

implementation. The software development life cycle, at the top-level, comprises five phases: requirements, analysis, design, implementation, testing and maintenance. Most approaches start by analyzing the system requirements based on use cases, which capture the expectations that the final users of the software may have. During the analysis phase, a complete specification of the system under development is established. In the subsequent architecture and design phases, a solution that provides the required services, i.e. fulfils the specification, is determined. Finally, environment-specific mapping strategies help developers to implement the design in a straightforward way for a specific platform. However, design methods, like HOOD and HRT-HOOD, focus more on the architectural and detailed design phases and leave the requirements analysis to the developer.

Software architectures, like DELTA-4, FRIENDS, and AQuA, however, do not offer any methodological support, but instead provide a structure (usually hardware design) based on which applications can be built.

Middleware is software that connects other applications to enable data flow between two possibly heterogeneous systems in a distributed computing environment. DCE, DCOM, Java RMI, and CORBA are examples of middleware. A *framework* is an environment composed of software components, which can be tailored according to the needs of the application being developed. Users deal with interfaces only and hence implementation details of the components are abstracted. A framework with NFR support, TARDIS, is presented later. Finally, a *middleware framework* is a structure that offers users multiple middleware styles that can be customized for application as well as device constraints. Some middleware frameworks, TIRAN and DepAuDE, are discussed in this thesis.

1.3 Research Objectives

The goal of this master thesis is to investigate if special-purpose development methods, i.e. processes targeted at a certain type of system (e.g. real-time systems), have addressed the concern of fault tolerance at the early steps of software development, e.g. at the requirements engineering and analysis phases. It would be of interest to see if other

approaches, like frameworks or middleware, are currently available that provide a methodological support for integrating fault-tolerance. It would also be relevant to study which of the non-functional concerns, such as reliability, timeliness, and safety, were handled in the approaches.

In order to address these questions, current specialized software development processes have been investigated and summarized in this thesis. The projects presented here include specialized software development methods, fault tolerance frameworks, middleware, software architectures, and proposed approaches that integrate the concern of fault tolerance in standard practices like the Unified Modeling Language (UML) [UML2003] and the Catalysis process [RL2004]. The thesis shows to what extent dependability requirements are addressed in each of the approaches, which application environments and failure domain are covered by each, and what fault tolerance techniques, if any, have been incorporated into the process. Special emphasis has been put on methodological schemes offered in the approaches for specification of fault tolerance and other non-functional requirements. Finally, a comparison of the different approaches has been established.

It should be noted that this thesis covers major available fault tolerance approaches but unintentionally some small ones may not be mentioned.

1.4 Organization

This thesis is organized as follows.

Chapter 2 provides an overview of software fault tolerance. It defines the fault tolerance terminology that is used throughout this thesis.

The survey is divided into three chapters.

Chapter 3 discusses two software development methods specialized in the development of real-time systems: HOOD and HRT-HOOD. HOOD provides limited support for

concurrent execution and real-time applications. HRT-HOOD is an extension of HOOD and aims to produce dependable real-time systems.

Chapter 4 mainly discusses three fault tolerance frameworks: TIRAN, DepAuDE, and TARDIS. These frameworks have integrated the concern of fault-tolerance in the software development process. TARDIS, however, was a proposed project and not followed up with more concrete work. In addition, general middleware that address fault tolerance have been reviewed at the end of this chapter.

Chapter 5 presents some proposed and developed approaches that consider elements required to produce dependable systems during the early stages of software development or during the design phases.

Chapter 6 presents the results of the survey based on a few comparison criteria.

Finally, **Chapter 7** discusses some possible future work and **Chapter 8** concludes on this thesis work.

Chapter 2.

Fault Tolerance Background

Systems are developed to satisfy a set of requirements that meet a need. A requirement that is important in some systems is that they be highly dependable. Fault tolerance is a means of achieving dependability. **Fault-tolerant systems** aim to continue delivery of services despite the presence of hardware or software faults in the system.

There are three levels at which fault tolerance can be applied. Traditionally, fault tolerance has been used to compensate for faults in computing resources (hardware). By managing extra hardware resources, the computer subsystem increases its ability to continue operation. *Hardware fault tolerance* measures include redundant communications, replicated processors, additional memory, and redundant power/energy supplies. Hardware fault tolerance was particularly important in the early days of computing, when the time between machine failures was measured in minutes [NISA95].

A second level of fault tolerance recognizes that a fault tolerant hardware platform does not, in itself, guarantee high availability to the system user. It is still important to structure the computer software to compensate for faults such as changes in program or data structures due to transients or design errors. This is *software fault tolerance.* Mechanisms such as checkpoint/restart, recovery blocks and multiple-version programs are often used at this level [NISA95].

At a third level, the computer subsystem may provide functions that compensate for failures in other system facilities that are not computer-based. This is *system fault tolerance.* For example, software can detect and compensate for failures in sensors. Measures at this level are usually application-specific [NISA95].

This chapter focuses on software fault tolerance and discusses evolution of failures from faults and techniques that can be used to tolerate such faults. It is primarily meant to provide a brief background on fault tolerance. It introduces non-functional requirements in Section 2.1, and in particular discusses dependability. Section 2.2 presents Laprie's concept of dependability, and discusses some important dependability attributes [JL92]. Section 2.3 describes the dependability impairments - faults, errors, and failures. Section 2.4 concentrates on fault tolerance techniques and includes error detection and system recovery.

2.1 Non-functional Requirements

In software systems, requirements can be broadly categorized into two types: functional and non-functional. **Functional requirements** define the behavior of the system, that is, the functions and services expected of the system. **Non-functional requirements (NFR)** address the constraints and qualities. NFR introduce quality attributes to a system that must be satisfied for the system to function and provide service under all circumstances. This is particularly important for dependable/real-time/safety-critical systems. Attributes of common concern for such systems include dependability, fault tolerance, availability, reliability, maintainability, safety, and security. But the question remains whether these should be treated as non-functional requirements, since as a result of this classification, important aspects like fault tolerance, are not addressed during the analysis and design phases [JK2003].

Non-functional requirements are also referred to as *desirable attributes*, *non-behavioral requirements*, *design constraints*, *system interface requirements*, *user interface requirements*, *hardware characteristics*, *software quality*, or more colloquially as "--ilities and --ities" [MB2001]. The classic non-functional requirement knowledge-base defined by Lawrence Chung in [CN2000] is included in Appendix A.

Three non-functional requirements often mentioned are *timeliness*, *adaptability*, *quality-of-service*, and most of all *dependability*.

Timeliness requirements are of concern in systems where correctness is not only based on the results generated but also on the time when they are made available [FL93]. Timeliness can be related to *soft real-time* or *hard real-time* requirements (discussed in Chapter 3).

Adaptability requirements involve satisfying the need to remain functional even when medications are carried out in the system or environment. Adaptability is associated to *dynamic change management* (making system modifications without halting the system), and *mode changes* (shifting the system goals according to changes in the environment) [FL93].

Dependability requirements are discussed in details in this chapter, starting from the next section.

2.2 Dependability: The Purpose of Fault Tolerance

IFIP WG10.4 Definition [JL92]

Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability has several attributes, including

- **availability** — readiness for usage
- **reliability** — continuity of service
- **safety** — non-occurrence of catastrophic consequences on the environment
- **confidentiality** — non-occurrence of unauthorized disclosure of information
- **integrity** — non-occurrence of improper alterations of information
- **maintainability** — capability to undergo repairs and evolution

The dependability requirement varies with the target application, since an attribute can be essential for one environment and not so much for others. The dependability tree is shown in Figure 1.

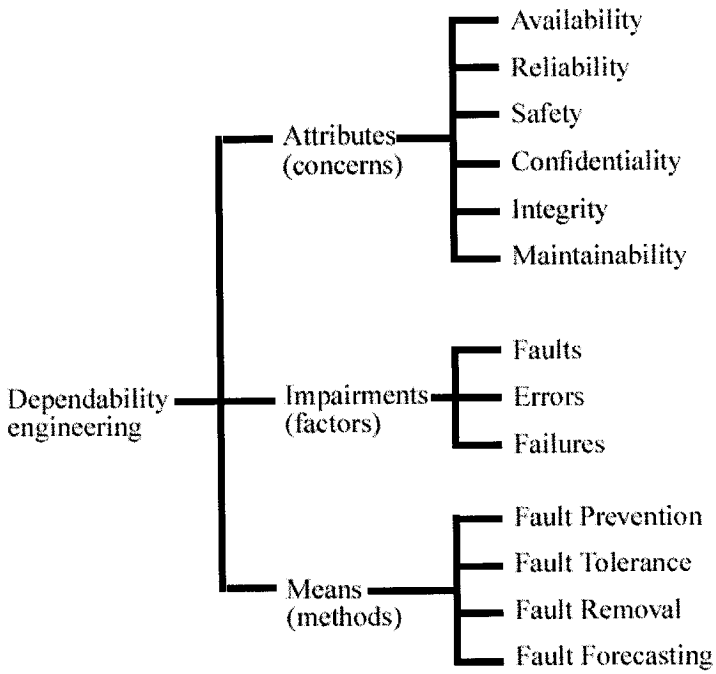

Figure 1. Dependability tree [JL92]

2.2.1 Reliability

The reliability of a system measures its aptitude to provide service and remain operating as long as required. The MTTF (mean time to failure) of a system is a quantitative measure of its reliability [GM2002].

2.2.2 Availability

The availability of a system measures its ability to provide the expected service whenever required. It can be calculated as MTTF/(MTTF + MTTR), where MTTF is the mean time to failure and MTTR is the mean time to repair [GM2002].

2.2.3 Safety

The safety of a system is determined by the lack of catastrophic failures it undergoes. It can be measured in binary terms, as in, a system can be safe or unsafe [GM2002].

2.2.4 Confidentiality

Confidentiality is the non-occurrence of unauthorized disclosure of information. The resource necessary to keep the information from being leaked out of the system is a measure of the strength of confidentiality [GM2002]. This is a security concern.

2.2.5 Integrity

Integrity is the non-occurrence of the improper alteration of information. It is related to the mean time to failure of a system and may be measured by the time and resources required for accessing a system without authorization. Integrity and confidentiality can be grouped as the security attribute.

2.2.6 Maintainability

The maintainability of a system is its ability to avoid, detect, localize, and correct faults. It can be measured as the MTTR (mean time to repair) together with consideration of the repair philosophy. To achieve maintainability, fault tolerance means can be utilized.

2.3 Dependability Impairments

Dependability impairments include aspects of a system, which cause a deviation from the normal behavior and entail loss of quality of service to some extent.

The IEEE TC FTD/IFIP WG10.4 definitions for the three impairments are as follows.

- A system **failure** occurs when the delivered service deviates from fulfilling the system function, the latter being what the system is aimed at.
- An **error** is that part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred.
- The adjudged or hypothesized cause of an error is a **fault**.

When the system delivers services which deviate from the correct behavior of a system according to its specifications, a failure is said to occur. Such a failure is caused by a

system state not adhering to the normal state and is called an error. The detection of an error implies the presence of a fault, also known as a bug, in the system. This cause-effect relationship creates a *fault->error->failure (FEF) chain,* which is an iterative process [TIRAN D1.1].

It should be noted that errors do not necessarily lead to failures; component failures are not necessarily faults to the surrounding system.

2.3.1 The FEF Chain

The FEF lifecycle includes three stages as in explained in [GM2002].

- The fault is dormant or **passive** when it is present in the system but has not yet lead to any errors or abnormal behavior.
- The fault is **active** when it effects the system functioning and hence produces an error.
- The error is propagated inside the system and results in a failure leading to a loss of service in the system.

The process by which a fault transforms into an error is known as **fault activation**. The spreading of errors that leads to a failure is the **error propagation** mechanism.

The concepts of *latency* and *inertia* can be illustrated in such a situation. *Latency* is the meantime between the occurrence of a fault and its activation as an error. *Inertia* is the meantime between the occurrence of a failure and the initiation of consequences to the environment [GM2002].

The following sections review faults, errors, and failures in more detail.

2.3.2 Faults

There are many types of faults that affect computerized systems, and they can be classified in various ways as shown in Figure 2.

Causes of Faults

- **Physical faults** can be caused by physical or environment phenomenon, like lighting.
- **Human-made faults** are functional or conceptual faults that are caused by humans during system development (e.g., bad system design) or improper handling of the system.

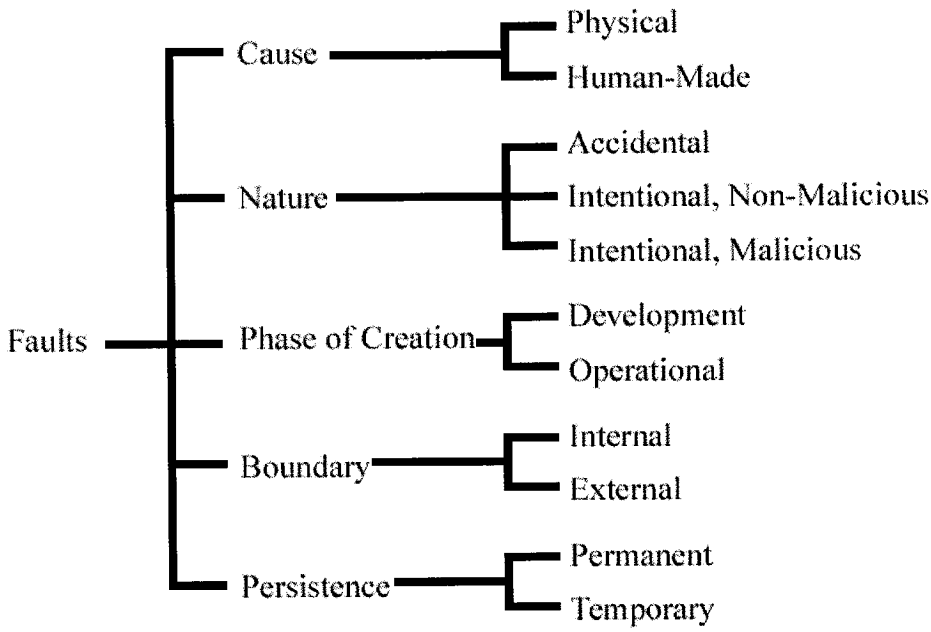

Figure 2. Fault classes [JL98]

Nature of Faults

- **Accidental faults** are more common and appear by chance because of environmental or human-made reasons. For example, a developer can misread a system's specification leading to bad analysis and design.
- **Intentional non-malicious faults** are due to compromises introduced during system design (e.g., not considering fault detection means exhaustively during design).
- **Intentional malicious faults** are those created deliberately by people. For example, one may spread viruses to sabotage the system.

Creation Phase of Faults

- **Development faults** are caused during system development. For example, choosing a bounded area that is too small during implementation might lead to a fault later.
- **Operational faults** crop up when the system is in operation. For example, activating parts of a system in the wrong order might case faults in the system later.

Boundary of Faults

- **Internal fault** are due to internal system states (e.g., a robot arm might malfunction because of incorrect logic used in the program).
- **External faults** are caused by environmental conditions or humans (e.g., excessive radiation might damage an equipment).

Fault Persistency

- **Permanent faults** or static faults appear and persist leading to a loss of service until the fault is detected and removed (e.g., extreme temperature damages an equipment).
- **Temporary faults** or dynamic faults disappear over time. Temporary faults can be transient or intermittent. *Transient faults* are caused by external events, like a loose power connection might make an equipment unusable for a while. *Intermittent faults* arise from internal reasons, like conflicting packets that may lead to temporary network partitions.

2.3.3 Errors

An error, an abnormal part of system state, does not necessarily cause a failure. This transformation is based on the level of redundancy in the system, the nature of the error (it might be temporary), and also whether it falls outside acceptable bounds (e.g., if it exceeds the acceptable error rate).

An error is said to be *latent* when it has not been discovered yet and subsequently it is *detected.*

2.3.4 Failures

According to Laprie, the *failures modes* of a system can be divided into three categories: domain, perception by several users, and consequences on the environment as shown in Figure 3.

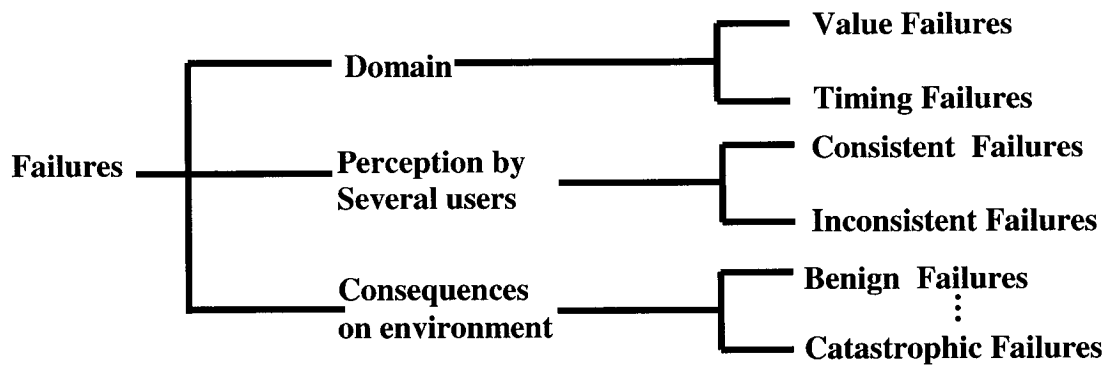

Figure 3. Failure classes [TIRAN D1.1]

Domain Failure

- **Value failure** occurs when an invalid value is output with reference to the correct behavior of the system. Value failure is a type of domain failure.
- **Timing failure** occurs when a service is provided before or after time. Late timing failures are also referred to as *performance failures*. When there is total loss of service, the failure is said to be a *halting failure*. Omission failures (failures that occur when the system does not respond to a request) and crash failures (failures that occur when the system stops responding completely) are classes of halting failure. A system that only experiences halting failures is said to be a *fail-halt* system. When a system fails and stops generating any output, it is said to be a *fail-passive* system. A system is said to be *fail-silent* when clients are not aware of the failure. Systems that have means to detect failures are known as fail-stop systems.

Perception by Several Users

- **Consistent failure**, a type of perception failure, is a failure which can be identified by all users in the same way.

- **Inconsistent failure**, a type of perception failure, is a failure which crops up in different ways for different users.

Consequences on Environment

The seriousness of the consequences of the failure can range from benign to catastrophic. Benign failures do not have catastrophic consequences on the environment. A system that only fails in a benign manner is said to be a *fail-safe* system. Catastrophic failures have disastrous consequences on the environment.

The risk and safety evaluation are application-dependent. The civil aeronautics defined a standard DO-178B to qualitatively evaluate the effects of failures [GM2002].

- *Without effects.*
- *Minor* or *benign* failures – lead to upsetting the passengers thus increasing the crew workload.
- *Major* or *significant* failures – lead to injuries of passengers and crew members thus reducing the efficiency of the crew.
- *Dangerous* or *serious* failures – lead to some casualties and/or serious injuries of passengers and members of the crew, or prevent the crew from achieving its task in a precise and complete manner.
- *Catastrophic* or *disastrous* failures – lead to loss of human lives.

2.4 Dependability Means

The dependability means are broadly categorized into four areas: fault prevention, fault tolerance, fault removal, and fault forecasting. These techniques are all related and should be considered together during development.

Fault prevention and fault removal means aim to increase the reliability or the availability (in case of repairable systems) of systems. Fault forecasting means are used to predict reliability of software. But together with these, fault tolerance techniques are required to

ensure acceptable levels of safety, reliability, and availability to take care of the residual faults.

2.4.1 Fault Prevention

Fault prevention involves taking measures to minimize the creation of faults and hence avoiding them in the first place. There are two tasks that need to be considered to reduce occurrence of faults.

- Measures need to be taken to act on the faults created by humans during the system development lifecycle.
- Measures need to be taken to account for faults that originate due to degradation or damage of the technology used.

Techniques used for fault avoidance/prevention include [LP2001]:

- refinement of the user's requirements iteratively – there should be communication between the software engineer and the system engineer to avoid faults that are created due to a bad specification,
- engineering of the software specification process,
- use of structured design and good software programming discipline,
- unambiguous coding,
- formal methods, and
- software reuse.

Fault prevention techniques aim to produce a system without any faults. But, this ideal scenario cannot be achieved usually, and it is important to consider the faults that might still exist in the system. This is where fault tolerance and fault removal means should be applied.

2.4.2 Fault Removal

Fault removal involves detection, diagnosis and removal of faults. This is considered at the software verification and validation phase.

- **Fault detection** tests for the presence of faults.
- **Fault localization** involves fault diagnosis and fault isolation and it identifies the fault present in the system.
- **Fault correction** eliminates faults when possible.

Fault removal means include:

- Software testing, a dynamic analysis technique, is commonly used fault removal. But, presently exhaustive testing is not possible and hence new faults are likely to crop up during the operation phase.
- Formal inspection is carried out before testing and entails checking the source code for faults. The errors are then corrected and verified.
- Formal design proofs are mathematical means used for removal of faults.

2.4.3 Fault Forecasting

Fault forecasting involves estimating the presence of faults in software and analyzing possible consequences of the faults [AL2001].

Fault forecasting techniques include:

- Reliability estimation uses inference techniques on the failure statistics generated during tests and operation to estimate the current reliability of the software [LP2001].
- Reliability prediction uses software metrics and measures during software development to estimate the reliability of the software in the future.

Fault removal and fault forecasting methods determine whether the software behavior is consistent with the specifications, but they cannot detect flaws in the specifications. Even

using the best people, practices, and tools cannot guarantee that the software is free of errors. Hence, fault tolerance means should be used to tolerate unexpected faults.

2.4.4 Fault Tolerance

Fault tolerance is the means by which a system can provide services in spite of faults, errors, or failures.

According to [AL2001], fault tolerance can be divided into two steps: error detection and system recovery as shown in Figure 4. Error detection means can be either concurrent or preemptive. System recovery comprises of error handling followed by fault handling.

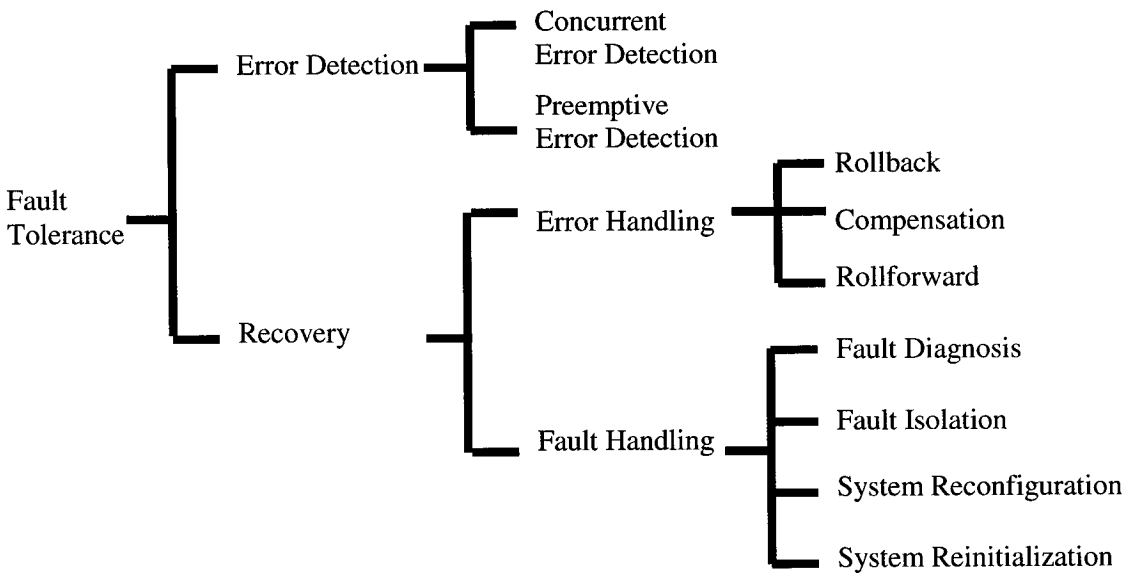

Figure 4. Fault tolerance tree

2.4.4.1 Error Detection Step

This step involves identification of errors in the system and uses forms of active redundancy for this purpose. There are two types of error detection mechanisms. Concurrent error detection means are used during service delivery. Preemptive error detection can only be done when the system is out of service or suspended, and is aimed at detecting latent errors and dormant faults. Some common error detection mechanisms are discussed in Table 1.

Objective	Mechanism
Corrupted messages	Cyclic redundancy code, parity checking, checksums, loopback testing
Late or lost messages	Time-out
Messages in disorder	Checking packet numbers
Late or dead processes	Watchdogs
Corrupted processes	Plausibility/range/history/authority checks
Corrupted environment	Control flow monitoring
Runtime errors	Exception processing
Memory violation, stack overflows	Memory access checking
Bit errors	Error detecting codes, parity bits, write/read back cycles
Periodic tests	Testing communication status, memory scrubbing, monitoring of error history

Table 1. Error types and detection mechanisms

Corrupted messages detection:

- *Coding checks* use redundancy in the representation of objects to help in the detection of errors associated to software or hardware.
  - *Cyclic Redundancy Code (CRC)* checks are used to detect errors in blocks of data related to hardware faults.
  - *Parity code*, the most popular and simplest code, involves adding one bit to a set of bits which is either 0 or 1 based on the XOR function of all the bits. For example, 1011011 has the parity bit 1 added to it. Multiple parity codes are used also.
  - *Longitudinal redundancy checks* add an extra bit to the end of every word in the block to be coded. V*ertical redundancy checks* add redundant words to the block. Both these checks together constitute the *bidimensional code check.*
  - *Unidirectional codes* detect all errors that alter the number of 1's in any one direction. For example for a word 101100, 101111 and 110000 are unidirectional errors. Multidirectional codes are also used for detecting errors.
- *Arithmetic codes* are used in calculation systems (addition, subtraction, multiplication, division) to detect arithmetic errors.
  - *Checksums* are used to detect errors in blocks of data that are caused by software-related faults. A checksum digit is sent with each packet which is a sum of the sequence of binary integers in the data.

- *Loopback testing* is used to check whether RS-232 communications hardware is in order. Data is sent from a port to a jumper and then back to the port. If the port, device, and cables are working, a key pressed appears on the screen.

Late or lost messages detection:

- *Time-out* helps in detecting communication errors caused by late or lost messages. A wait period for a message is set and on exceeding this, an alert is raised or measures are taken to handle it.

Disordered messages detection:

- *Packet sequence numbers* is a number added to each packet that helps the receiver detect errors by checking whether any one is missing.

Late or dead processes detection:

- *Watchdog* mechanisms can be used with data diversity and design diversity techniques to detect late or dead processes. Watchdog timers set a deadline for acceptance tests to run on primary algorithm and when the deadline expires a backup algorithm maybe invoked.

Corrupted processes detection:

- *Plausibility checks* are used to check whether a value falls in the acceptable range (limit range, deviation from a default value range, or results from a pattern check)
- *Range checks* are consistency checks that confirm whether a computed value is in a valid range, for example, a computed probability must be between 0 and 1.
- *Address checking* verifies that the address to be accessed exists.
- Other checks to detect corrupted processes include *history checks* and *authority checks.*

Corrupted environments detection:

- *Control flow monitoring* (or *Control Flow checking)* entails partitioning the application program in basic blocks, i.e., branch-free parts of code. For each block a

deterministic signature is computed and faults can be detected by comparing the run-time signature with a pre-computed one.

Run-time errors detection:

- *Exception processing* is an online error detection mechanism which signals the presence of errors during program execution by raising exceptions.

Memory violation/stack overflow detection:

- *Memory access checking* is required to detect errors caused by reading from uninitialized memory or writing to freed memory, which can cause a program to behave abnormally or even crash. Compilers and tools need to be used to find statically-detected and runtime-detected errors.

Bit errors detection:

- *Parity bits* are used for parity checking (explained above).
- *Write/read back cycles* are used to prevent unintentional storage of bad data. A write operation is followed by a read operation to detect bad recording of data.

Detection via periodic tests:

- *Memory scrubbing* merely reads out data to a controller, scrubs out any correctable error(s), and writes the data back into memory before multi-bit errors build up and become no longer correctable.
- *Monitoring of error history* involves keeping a trace of all runs and analyzing them to detect error patterns.

Two terms often associated with error detection are *error diagnosis* and *error isolation.* Error diagnosis involves determining the causes of the error and assessing the damage to the system. After detection, error isolation or active confinement mechanisms are necessary to isolate the erroneous component from the other parts of the system to prevent error propagation. This might involve termination of a faulty communication channel, termination of a faulty process, or disconnection of a faulty node.

2.4.4.2 System Recovery Step

This step involves correcting and/or repairing the errors and faults, so as to regain an error-free or fault-free system. Recovery might involve retransmission of messages, re-initialization or starting of new communication channels, re-initialization or starting of new processes, re-initialization of a node, or checkpointing and rollbacks.

2.4.4.2.1 Error Handling

This step aims to remove errors from the system state using rollback, rollforward, or compensation mechanisms.

- **Rollback** or **backward error recovery** is used to restore the system to an earlier error-free state. It is hence necessary to create recovery points (or checkpoints) that save the current system state at predetermined intervals to stable storage (a storage whose contents survive assumed failures) such that the error-free state can be used later for rollbacks. This mechanism helps in recovering from unexpected errors and damages and is particularly suitable for recovery from transient faults. Drawbacks of this approach include requirement of significant resources, temporary halt of the system, and possible occurrence of the domino effect.
- **Compensation** involves use of redundancy to mask an error by only selecting an acceptable result based on some algorithm, thus making it possible to transform to an error-free state. Modular redundancy along with majority voting is a common technique to achieve compensation.
- **Rollforward** or forward error recovery involves restoration of the system to a new state which may be a degraded one. But this requires knowledge of the errors and hence is application-specific. This approach is however efficient and suitable in cases of anticipated faults and missed deadlines.

2.4.4.2.2 Fault Handling

Fault handling aims to ensure that faults are not activated again.

2.4.4.2.2.1 Fault Diagnosis

Fault diagnosis involves determining the cause and class of an error. This is not the same as error diagnosis since different faults can lead to the same error.

2.4.4.2.2.2 Fault Isolation

Fault isolation or fault passivation is used to prevent re-activation of a fault. This process involves fault masking and fault containment.

Fault masking is the means by which a fault is corrected and kept hidden from the system boundary, that is, it is masked transparently. Techniques used include auto-correcting code, N-modular redundancy, etc.

Fault containment (or *passive confinement*) is the process by which fault propagation can be prevented such that a fault is not allowed to lead to loss of service. The system is structured in fault confinement regions such that one region communicates with another by means of carefully monitored messages.

Fault compensation maybe necessary after fault containment to cover for the lack of output of the faulty component.

If fault isolation leads to a loss of service, fault reconfiguration needs to be considered.

2.4.4.2.2.3 System Reconfiguration

System reconfiguration measures are used when faults appear, to alter the system configuration such that the non-failed redundant components can takeover execution of the system with some possible degradation [TIRAN D1.1].

2.4.4.2.2.4 System Reinitialization

After reconfiguration, it is necessary to check, update, and record the new configuration and update the system tables and records [AL2001].

2.4.4.3 Fault-tolerance Mechanisms

Fault tolerance techniques maybe provided for single version software environments or multiple version software environments. Such techniques have been employed in many

fields including aerospace, healthcare, telecommunications, nuclear power, and ground transportation industries [LP2001].

Single version software environments

The following are some common measures taken to tolerate software faults.

- Monitoring techniques are used for online tests during system operation. The functioning of the system is observed and signals are sent when erroneous or impending faulty behavior occurs. This makes diagnosis easier during the system maintenance. For example, a light signal is activated when a car is out of gas [GM2002]. Watchdog timers (discussed earlier) are a type of monitoring technique.
- Atomicity of actions requires that the actions of one group of components do not interact with other groups during the time the activity is being executed. This ensures that the state alterations made by the action are recorded correctly when no failures occur. In case of a failure, the changes are not committed and the system rolls back to the state prior to the execution of the action.
- Exception handling is a way of handling errors that might appear in a component. This should be considered in the design phase and the system should be implemented such that abnormal conditions can be handled or an exception is raised as a signal of an error. Exception handling is often considered along with *idealized fault-tolerant components (IFTC).*
- A software system designed using the IFTC approach [LA90][RX95] is split into a set of components with well-defined interfaces and boundaries as shown in Figure 5. A component can be requested to perform a service. During normal processing the component performs the service, possibly by calling services of other components, and returns the result. If a request is malformed, an *interface exception* is raised. In the case when a local exception occurs during processing of a request, the component can itself try to address the error, if successful, the component returns to normal processing. But when the exception cannot be handled locally, a *failure exception* is raised and propagated to the request of the service.

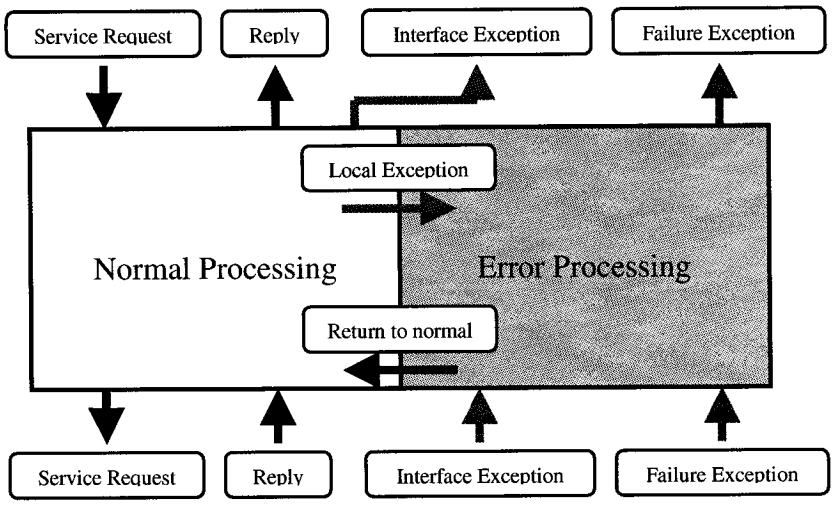

Figure 5. Idealized fault-tolerant component

Multiple version software environments

Design diversity and data diversity methods are used to tolerate faults in multi-version environments. The methods can be static or dynamic.

Dynamic software fault tolerance techniques entail selecting the result of one variant as the acceptable output. This decision is made during program execution based on an acceptance test (an adjudicator which determines whether system behavior is acceptable) [LP2001].

In *static software fault tolerance techniques*, several versions of a program execute concurrently, in separate processes or separate processors, and following that a result is accepted or determined using some decision mechanism [LP2001].

Design diversity techniques involve redundancy of software or hardware systems.

- **Recovery blocks (RB)** [HL74][BR75] are a dynamic technique to detect and tolerate software faults. They use acceptance testing (AT) and backward error recovery. The functionality is implemented using different algorithms all doing the same work. The most efficient algorithm is taken as the primary and is executed. If the acceptance test does not pass for this algorithm, the next alternate

is executed and this goes on till the AT passes or all the algorithms have been executed. On failure, an error is raised. 0 illustrates the RB technique. Variants of the recovery block include distributed recovery blocks and consensus recovery blocks.

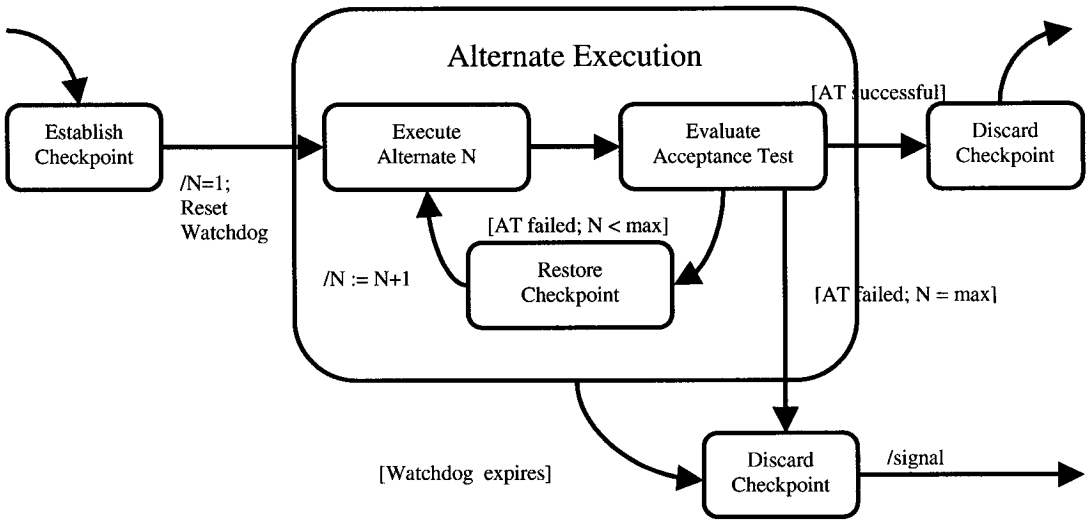

Figure 6. Recovery block structure and execution [JK2003]

- **N-version programming (NVP)** [WE72][CA78] is a static technique which uses a decision mechanism (DM) and forward error recovery. At least two versions providing the same functionality are designed independently and executed concurrently. The DM accepts the correct or "best" result. The DM is usually based on a majority voting method (details are not mentioned here). Other variants of NVP like N-self checking programming are available. Figure 7 illustrates the NVP technique.

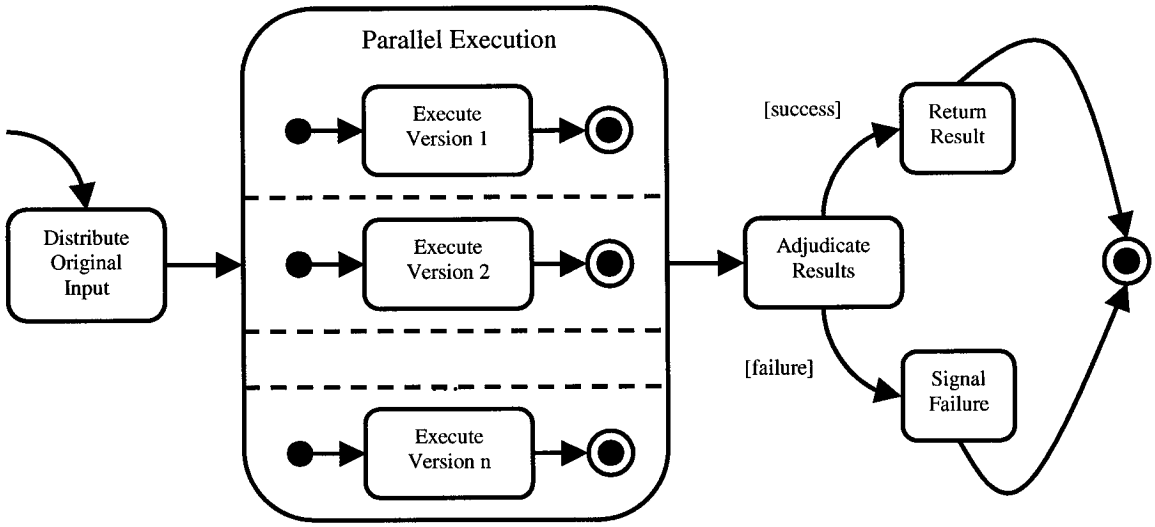

Figure 7. NVP structure and execution [JK2003]

Data diversity means involve representation of the input data in different ways so as to effectively handle faults in environments using multiple data representation.

- **Retry blocks** [AK87] is a dynamic technique which runs on a sequential environment and uses an acceptance test, a data re-expression algorithm, a primary algorithm, a watchdog timer, and a backup algorithm. The primary algorithm is first run on the original input and if the acceptance test does not pass, the input is re-expressed using a data re-expression algorithm (DRA) and the primary algorithm is executed again. This continues until the watchdog expires, after which the backup algorithm is executed on the original data. On failure, an exception is raised.
- **N-copy programming (NCP)** [AK88] is a static technique which uses a decision mechanism (DM) and forward recovery to achieve fault tolerance. The original input is put through data re-expression algorithms (DRA) and then passed onto variants of the same process. The DM then examines the results and selects the best one, if one exists.

Chapter 3.

Software Development Methods

This chapter discusses two software development methods: HOOD [HRM4] and HRT-HOOD [BW95]. Section 3.1 focuses on the software design methodology used in HOOD and also includes suggested requirements analysis techniques. Section 3.2 discusses HRT-HOOD which is a HOOD based method that uses the abstractions defined in HOOD along with new additions to cope with hard real-time issues. This section of the chapter concentrates on the architectural design steps followed in HRT-HOOD (and also HOOD). At the end of the chapter, a case study of the mine control system is presented. Other projects related or based on HOOD and HRT-HOOD are briefly mentioned as well.

3.1 HOOD

HOOD (Hierarchical Object-Oriented Design) [HRM4] is an architectural design method developed as an ESA (European Space Agency) project in 1987. The method is primarily intended to produce source code in Ada. The Ada language was chosen by ESA since it specifically catered to embedded software and such software were a major part of ESA's projects.

HOOD is a diagrammatic object-based method. It helps in building hierarchies of objects that can be formalized into structured text and later refined into code.

3.1.1 Software Development Life Cycle

The HOOD method supports development of systems after requirement analysis activities to code generation. It provides extensive guidelines for architectural and detailed design, and also for mapping the design to of code and for testing.

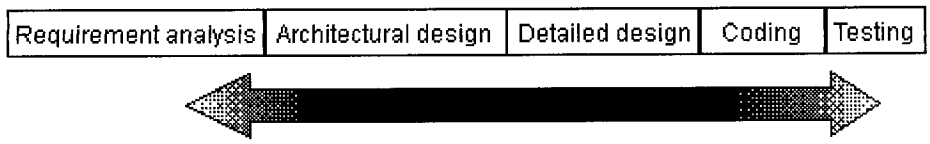

Figure 8. Scope of HOOD [JR97]

The HOOD method follows primarily a top-down approach. The guidelines provided take the form of a *Basic Design Step* which is used to produce hierarchies of objects and to transition from one development phase to another.

3.1.1.1 Basic Design Step

The HOOD method uses a *basic design step* which is aimed at identifying an object and its child objects, finding the association of the object with other objects, and finally to map terminal objects to code.

The Basic Design Step comprises four steps: problem definition, development of the solution strategy, formalization of the strategy, and formalization of the solution, as shown in Figure 9.

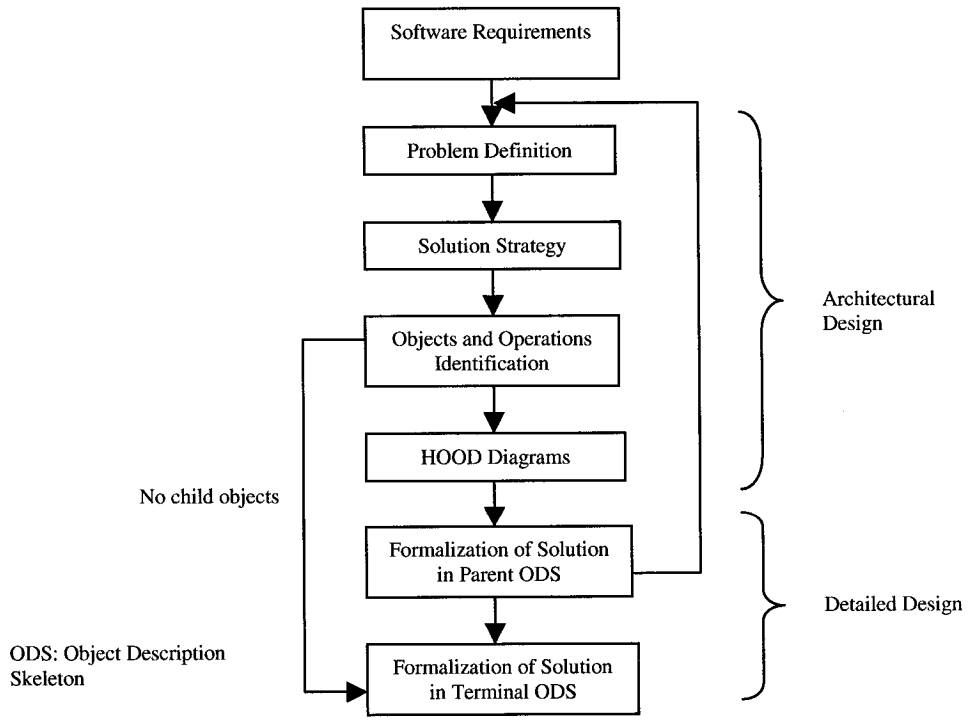

Figure 9. Basic design step

3.1.1.1.1 Problem Definition

The problem definition phase includes two steps: stating the problem and analyzing the requirements.

3.1.1.1.1.1 Statement of the Problem

The problem definition involves referring to a requirements document and if no such document is available then to state the problem in precise sentences.

3.1.1.1.1.2 Analysis and Structuring of Requirement Data

From the problem statement, the relevant information is collected and analyzed. The requirements can be categorized into the following three types.

- *Static functional requirements* are used to identify objects and define the tasks to be carried out by each object which are later reflected in the definitions of the operations during the formalization of the solution.
- *Dynamic functional requirements* are used to identify the type of object and describe the dynamic behavior of the system using Petri nets or state transition diagrams, which are later integrated in the formalization of the solution.
- *Non-functional requirements* are used to state constraints on the system. HOOD does not put much focus on the definition and design of non-functional requirements. They are included in one section in the formalized text solution but no concrete guidelines are provided as to how these should be designed and implemented. HRT-HOOD, discussed in Section 3.2 of this chapter, considers these requirements with respect to real-time systems to a much greater extent.

3.1.1.1.2 Development of the Solution Strategy

This phase involves outlining a solution for the problem defined by using a natural language. Initially, the solution provided is a top-level one and describes how the design will work. In later phases, this strategy is to be refined.

3.1.1.1.3 Formalization of the Strategy

In this phase, the objects and their associated operations are defined. Following this, the HOOD diagrams are produced and any design decision is justified.

3.1.1.1.3.1 Object Identification

The initial task in the HOOD method is to identify the objects. The identification process involves extracting nouns from the solution strategy. These nouns have to be structured by behavior and level of abstraction [PR92]. Each noun is then classified as a child object, an attribute of an object, a value of an attribute, or just irrelevant words. The list of useful nouns is then saved as the results of the *object identification* phase.

3.1.1.1.3.2 Operation Identification

The operation identification is similar to the object identification process. In this case, the verbs are identified from the solution strategy and examined to see if they are relevant to the level of abstraction. The properties associated to the execution, like parallelism, synchronization, periodic execution, need to be defined in this phase. The list of verbs and their properties are then saved as the *operation identification.*

3.1.1.1.3.3 Grouping Objects and Operations

In the third step, each operation is associated with an object. This results in an *object operation table* (OOT) which lists objects with their associated operations.

3.1.1.1.3.4 Graphical Description

A diagram is produced based on the HOOD graphical formalism. The object hierarchy (parent-child objects) is identified. Control flow dependencies which show how each object uses child objects, are added to the diagram. Also, major data flows and exception flows are added to express the relationships between objects. A mapping is added for each operation of a parent object that is implemented by an operation of a child object.

Details about the elements in a HOOD diagram and the steps used to produce the diagrams are not discussed here. Section 3.2.4.3 of this chapter describes the graphical formalism and along with extensions that are available in HRT-HOOD.

3.1.1.1.3.5 Justification of Design Decisions

It is suggested that the design decision be documented to aid in maintenance and future work of the system. It is required that decisions for objects (other than passive objects or objects representing the environment), constrained operations, and identified exceptions be justified.

3.1.1.1.4 Formalization of the Solution

This phase elaborates a formal model of the solution – the Object Description Skeleton (ODS). The ODS is a structured text format which formalizes the solution strategy by describing each object. The ODS will be the source of documentation for detailed design and code generation.

3.1.1.1.4.1 HOOD Tools

The formalization can be automated with the use of HOOD tools. Such tools are seen as necessary for software development using the HOOD method. They support development by providing editors, by generating documents in desired standards, by performing consistency checks of the representations, and by analyzing the design in various ways [JR97].

Several HOOD tools are available on the market from different vendors. The Standard Interchange Format defined by HOOD makes it possible for designs to be compatible across several tools [JR97].

3.1.2 Real-time Design in HOOD

HOOD supports two major models for real-time system design.

- Asynchronous model

In this model, the software has components at different priorities. High priority software is dedicated to providing quick response to external events, medium priority software do the normal processing, and low priority software do less essential tasks such as logging. HOOD provides active objects for concurrent execution, constrained operations to synchronize active objects, and the asynchronous execution request (ASER) to support interrupts.

- Synchronous model

In this model, the software is divided into components with allocated time-slices for each. During execution, each component is allowed to execute for a fixed period of time. Exceeding the time limit indicates serious design problems. In addition to all elements provided in the asynchronous model, HOOD provides the passive object implementation specifically for this model.

HOOD looks into some concerns in real-time systems and puts constraints on the design accordingly.

- Deadlocks – HOOD looks into the use of active objects and their interfaces from the initial design phases. Hence, potential deadlocks can be noticed early and actions taken to prevent them.
- Infinite Recursion – HOOD does not allow cyclic use of passive objects and recommends use of seniority hierarchy. This helps prevent infinite recursions from occurring.

However, the HOOD approach is not adequate to produce dependable real-time systems. It lacks "abstractions that directly relate to common hard real-time activities" [BW94]. These issues have been addressed in the HRT-HOOD method which is discussed later in this chapter.

3.1.3 Concurrency

HOOD supports concurrency by providing active objects which correspond to Ada tasks. For performance reasons it is, however, recommended that the number of active objects be kept at a minimum and transformed into passive objects as much as possible.

3.1.4 Benefits of using HOOD

The HOOD method has been used widely for design of space real-time systems and also in other domains including energy, defense, and transport. Some projects include the ESA Columbus Space Station, ESA Hermes Space Plane, European Fighter Aircraft, and Thorn EMI Electronics (design of a multi-processor real-time system). HOOD offers clarity of design and ease of extensibility, mapping to manpower, and integration. It is a public-domain method and produces maintainable and reusable software. HOOD also provides abstractions to support the concept of distribution in Ada software [HRM4]. Additionally, it is supported by various toolsets. HOOD is also compatible with the Unified Modeling Language (UML).

3.1.5 Limitations of HOOD

HOOD does not encompass all aspects of object-oriented design (e.g., inheritance, dynamic binding, etc.). Hence, the OO in HOOD can be misleading since it is more object-based than object-oriented.

HOOD does not provide any means to carry out requirements analysis and does not make any preconditions about the method to be used. It is, however, required that the design is preceded by object-oriented analysis to find the objects for design. The suggested technique discussed in Section 3.1.1.1.3 is quite cumbersome and error-prone.

HOOD does not suggest any methods for maintaining traceability of objects, operations, and other design information back to the source code.

3.1.6 Developments and Related Projects

The original object-based HOOD method has been upgraded with object-oriented concepts including classes, inheritance, and polymorphism. Also, initially HOOD supported code generation in Ada83 only. But, it has been adapted to support other languages like C and FORTRAN and even object-oriented languages like C++, Ada95,

and Eiffel. Moreover, other projects have been inspired by HOOD and have been developed as extensions of HOOD.

3.1.6.1 HOORA

Hierarchical Object Oriented Analysis (HOORA) [HOORA] is an ESA developed methodology that came about in March 1995 as a consequence of the HOOD method. HOORA is a method that supports requirements analysis of software and is truly based on the object-oriented paradigm. It uses the UML notation and can be used with languages like Ada and C++ to produce object-oriented based software. It provides concrete guidelines to transition to HOOD and ultimately to Ada source code. "HOORA not only offers a diagramming notation, but also a complete framework and process for describing and analyzing the static and dynamic behavior of software systems" [GG95]. Toolsets are available that support the HOORA method, and which can also automatically produce an initial HOOD design. Details about the HOORA method can be found at [HOORA].

3.1.6.2 HRT-HOOD

Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD) [BW95] is an ESA (European Space Agency) project, which is an extension of HOOD (Hierarchical Object-Oriented Design). HRT-HOOD is presented in the next section.

3.2 HRT-HOOD

Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD) is an ESA (European Space Agency) project developed jointly by British Aerospace Space Systems Ltd., The University of York, and York Software Engineering Ltd, during 1991-1993. This is an extension of the standard method used by ESA for architectural design, HOOD (Hierarchical Object-Oriented Design). HRT-HOOD includes extra features catered towards design of embedded real-time software. The method aims to support functional and timing correctness and also considers timing requirements early on during

development to avoid the usual trend of making comprises during detailed design and implementation.

3.2.1 Real-time Issues

3.2.1.1 Real-time Systems

A system which is constrained by non-functional requirements, esp. timing constraints, is said to be a real-time system.

3.2.1.1.1 SRT and HRT Systems

In **hard real-time (HRT) systems**, timely services are crucial. Failure to provide the service may lead to severe damage to the system or environment and loss of life. Examples of HRT systems include flight control systems, spacecraft control computers, and nuclear power plant control systems.

In **soft real-time (SRT) systems**, the response times are important but a few missed deadlines do not cause the system to behave badly or to stop functioning. Vending machines and ATMs are examples of SRT systems.

3.2.1.1.2 Non-functional requirements

Real-time systems have to meet additional non-functional requirements (NFR), such as dependability (reliability, availability, safety, security), timeliness (responsiveness, orderliness, freshness, temporal predictability, temporal controllability), and dynamic change management (incorporating evolutionary changes into a non-stop system). These requirements need to be considered during the software development life cycle.

3.2.1.2 Extensions in HRT-HOOD

HRT-HOOD addresses issues of timeliness and dependability in the early stages of the development process. Extending HOOD, it has explicit support for common hard real-time abstractions. There are few other methods which provide abstractions which can be directly associated to hard real-time activities. Although methods like HOOD and MASCOT [HS86] can be used to design hard real-time systems, they do not provide

means to analyze timing properties of systems and their use can lead to erroneous designs.

3.2.2 Software Development Life Cycle

The HRT-HOOD software development life cycle consists of the following activities.

- *Requirements definition* is concerned with specifying the functional and non-functional behavior that the system must adhere to.
- *Architectural design* involves producing a top-level design of the system being developed.
- *Detailed design* specifies the complete system design.
- *Coding* deals with the implementation of the system.
- *Testing* involves checking the effectiveness and efficiency of the system.

The architectural design comprises two activities: logical design and physical design. The logical design takes into account the functional requirements of the system. The physical architecture is a refinement of the logical one and considers these requirements along with the constraints imposed by the execution environment (the hard and software components) and aims to satisfy the non-functional requirements. The physical architecture ensures that the timing and dependability properties of the system are met.

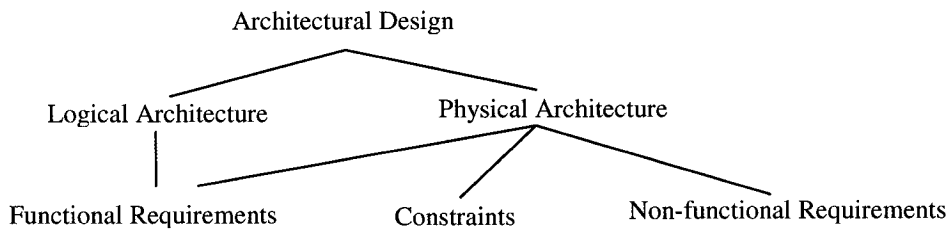

Figure 10. Scheme of the design phase

Following the architectural design, the detailed design is carried out which finally leads to the coding stage. The code must be checked to see whether the initial estimated worst-case execution time holds and any inconsistency requires the detailed or the architectural design to be revisited. On satisfactory results, the testing phase, which includes code

timing measurements can commence. The complete HRT-HOOD software development life cycle is shown in Figure 11.

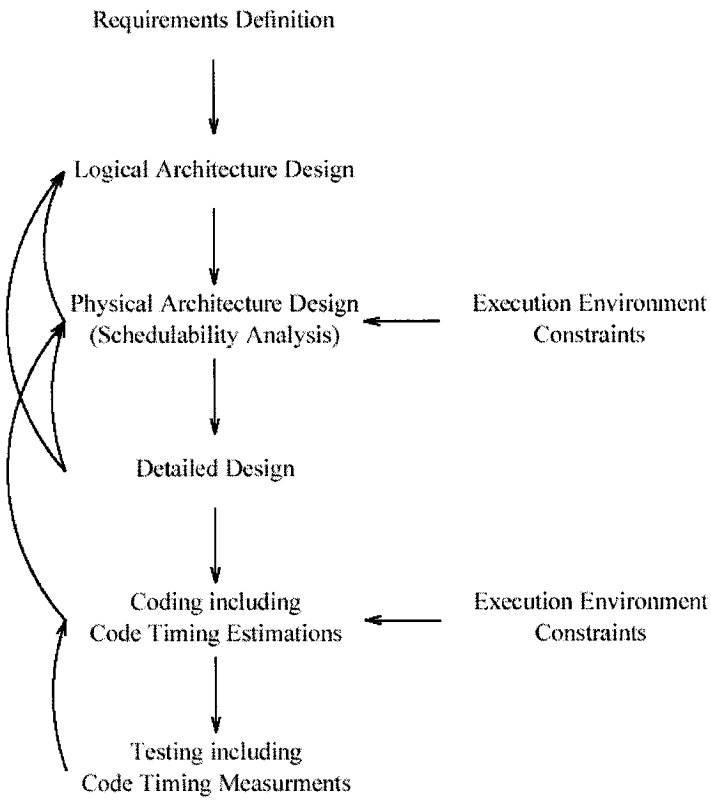

Figure 11. The HRT-HOOD software development life cycle [BW94]

3.2.3 Requirements Definition

HRT-HOOD does not define techniques for requirements specification and analysis. It relies on the means proposed in the HOOD methodology.

3.2.4 Architectural Design

3.2.4.1 Logical Design

HRT-HOOD provides clear guidelines to produce a structured logical design. The logical design involves creating abstractions which are specific to hard real-time systems, like support for periodic and sporadic activities. In addition, this step introduces constraints on the design that allow it to be analyzed later.

3.2.4.1.1 HRT-HOOD Objects

HRT-HOOD extends the two basic object types of HOOD. Along with objects of type *passive* and *active*, it includes *protected*, *cyclic* and *sporadic* objects. The objects are defined in section 3.2.4.3.

3.2.4.1.2 Design Constraints

There are some constraints that HRT-HOOD imposes on the design. Most of these are constraints on the communication and synchronization between objects. Such constraints are important for producing analyzable software. In the case of HRT-HOOD, one of the goals of the method is to provide a framework that produces software that can be analyzed for its timing properties. The design constraints include the following.

- Cyclic and sporadic objects should not call arbitrary blocking operations in other cyclic or sporadic operations but are allowed to call operations with an asynchronous transfer of request.
- Protected objects should not call blocking operations in any object.
- Passive objects do not have to be considered for synchronization.

3.2.4.2 Physical Design

The physical design of the system primarily involves adding annotation to objects defined in the logical architecture. This is done by the use of object attributes. This phase also provides support for carrying out schedulability analysis of terminal objects and provides the abstractions necessary to "express the handling of timing errors" [BW94].

3.2.4.2.1 Object Attributes

Non-functional requirements are addressed in the HRT-HOOD method by assigning several attributes to each object.

- Cyclic object: {Period of execution, Offset times, Deadlines}
- Sporadic objects: {Minimum arrival time, Offset times, Deadlines}

Deadlines can be set directly to cyclic or sporadic activities or to transactions (precedence constrained activities). The worst-case execution time needs to be monitored for each

thread to enable schedulability analysis. Some scheduling algorithms require a precedence level to be set. In addition, processes are set to a criticality level – safety critical, mission critical, background. This level is considered during validation and verification.

The physical design can proceed along with the logical design. Objects added at the logical level may need to be assigned timing attributes. It is also possible to add objects at the physical architecture level. This might be done to support replication or for the reduction of output jitter[2] (a non-functional requirement). The latter can be achieved by object decomposition enabling objects to execute tasks in parallel and hence reduce the jitter.

3.2.4.2.2 Tolerating timing faults and errors

It is important to monitor the system for timing errors, so that they do not lead to failures. The designer should make sure that the object does not use more than its share of computation time and that it does not execute after its specified deadline. In cases of timing faults, an exception should be raised and the object should be able to handle this. The coding language should have support available to program recovery handling. In cases of sporadic objects, method invocation should be monitored in order to prevent early execution or overly high invocation frequency.

3.2.4.3 Hard Real-Time HOOD Objects

The logical architecture phase results in a collection of terminal objects. This section defines the objects types used in HRT-HOOD (both terminal and non-terminal). Along with the two object types defined in HOOD (active and passive), HRT-HOOD adds three objects to the design: protected object, cyclic object, and sporadic object. The objects are graphically represented as shown in Figure 12.

[2] "Jitter is the deviation in or displacement of some aspect of the pulses in a high-frequency digital signal" [SN].

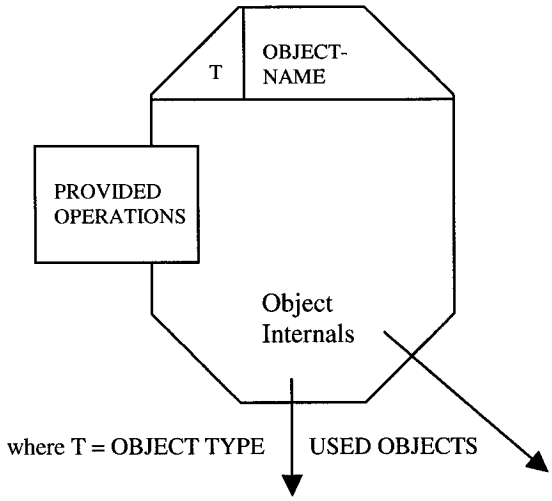

Figure 12. Graphical representation of an HRT-HOOD object [BW94]

3.2.4.3.1 Passive Objects

Passive objects as the name implies are passive and do not restrict their operations from being executed. When an operation of a passive object is called, the operation gets immediate control. The instructions are executed sequentially and the object does not synchronize with other objects.

3.2.4.3.2 Active Objects

Similar to passive objects, active objects can be unconstrained. But, active objects are usually used to specify constraints on the execution of its operations.

The constraints can be of two types: *functional activation constraints* (optional) and *types of request* (compulsory).

- Functional Activation Constraints

This type of constraint restricts execution of operations unless the object's state allows it. If the execution is allowed, the operation is defined as "open", otherwise it is "closed". By default, an operation is "open".

- Types of Request

The way an operation is constrained is defined in a label of the trigger arrow in the HOOD diagram.

Asynchronous Execution Request (ASER) – Operations called with this request type do not require the invoker to be blocked.

Loosely Synchronous Execution Request (LSER) – Operations with LSER request types require the invoker to be blocked until the invokee is ready to execute the operation.

Highly Synchronous Execution Request (HSER) - In this case, the invoker is not allowed to execute further until the request is serviced.

Time Operation Execution Request (TOER) – Requests with TOER have a time-out period specified along with the request. Request types include TOER_LSER and TOER_HSER.

3.2.4.3.3 Protected Objects

Protected objects are required when resources accessed by hard real-time systems need to be monitored and their access limited. This is achieved by restricting invocation of operations. The use of protected objects enables "the run-time blocking for resources to be bounded" [BW94]. These objects do not necessarily require independent threads of control.

Protected objects can have two types of constrained operations: PSER (Protected Synchronous Execution Request) and PAER (Protected Asynchronous Execution Request). These operations can only execute in mutual exclusion. Protected objects can also have unconstrained operations that behave similar to passive objects.

3.2.4.3.4 Cyclic Objects

Cyclic objects are required to denote periodic activities. They are active objects but they do not suspend execution regardless of "outstanding requests for their objects' operations" [BW94].

Cyclic objects do not include operations usually but at times for efficiency reasons, operations can be defined in cyclic objects. These operations are all of asynchronous transfer of control request type (ATC). The operations can be further defined as asynchronous ATC (ASATC), loosely synchronous ATC (LSATC), and highly synchronous ATC (HSATC).

3.2.4.3.5 Sporadic Objects

Sporadic objects are active. They contain an independent thread of control which is activated by invoking a "start" operation defined in the object. Objects can be simply of type ASER (asynchronous execution request) or if invoked by an interrupt, ASER_BY_IT. ATC (asynchronous transfer of control) request type can also be defined on sporadic objects, in which case, the sporadic object halts its current operation.

3.2.4.3.6 Environment Objects

An environment object is used to incorporate other software into the design without disrupting with the HRT-HOOD principles. It is represented with the letter E as the object type.

3.2.4.3.7 Objects Summary

	Unconstrained Operation	Constrained Operation	Functional Activation Constraints	Type of Request	Timeout
Passive	✓	×	×	×	×
Active	✓	✓	Optional	✓	✓
ASER	×	✓	Optional	✓	×
LSER	×	✓	Optional	✓	✓
HSER	×	✓	Optional	✓	✓
Protected	✓	✓	Optional	✓	✓
PSER	×	✓	✓ (if TOER)	✓	✓
PAER	×	✓	×	✓	×
Cyclic	×	✓	Optional	✓	×
ASATC	×	✓	Optional	✓	×
LSATC	×	✓	Optional	✓	×
HSATC	×	✓	Optional	✓	×
Sporadic	×	✓	×	✓	×
ASER	×	✓	×	✓	×
ASER_BY_IT	×	✓	×	✓	×
ATC	×	✓	×	✓	×

Table 2. HRT-HOOD objects

3.2.4.4 HRT-HOOD Abstractions

3.2.4.4.1 HRT-HOOD Object Attributes

HRT-HOOD as opposed to standard HOOD introduces real time attributes of objects which are used to define real-time constraints.

- Deadline – This attribute specifies the execution deadline of cyclic and sporadic objects, if required.
- Operation_Budget – A budget execution time can be specified for each externally visible operation. In case this budget is exceeded, an internal error handling operation must be available in the object.
- Operation_WCET – This is required for all operations used for interfacing to specify the worst-case execution time (WCET) of operations. The WCET of an operation is the sum of the Operation_Budget and the budget time of the internal error handling operation.
- Thread_WCET – A cyclic or sporadic object must specify a worst-case execution time for its thread. The WCET is the sum of the thread_budget and the internal error handling operation budget time.
- Thread_Budget – A cyclic or sporadic object can have a budget time specified for its thread. Similar to the operation-budget, if the thread budget is exceeded, then an internal error handling operation should be available.
- Period – This attribute is provided to specify the period of execution of cyclic objects.
- Offset – An offset time can be added to cyclic objects which would specify the waiting time before commencing its operations.
- Minimum_Arrival_Time or Maximum_Arrival_Frequency – These attributes are added to sporadic objects to specify the minimum arrival time or the maximum arrival frequency for requests of its execution. At least one of the two must be specified.
- Precedence Constraints – This is used to specify the precedence order of threads.
- Priority – The scheduling theory used may need a priority to be set for threads of cyclic and sporadic objects.

- Execution_Transformation – It may be required to transform cyclic and sporadic objects during execution to add extra delays.
- Importance – This attribute is used to specify whether a thread is hard real-time or soft real-time.

Depending on the application, attributes for minimum/average execution times, utility/benefit functions, replication, integrity levels may be added.

3.2.4.4.2 The Relationships and Rules

3.2.4.4.2.1 Use Relationship

The USE relationship in HRT-HOOD follows the following rules.

- **Passive objects** can only use unconstrained operation of other objects. Passive objects are not allowed to make cyclic use of each other.
- **Active objects** are allowed to use any operation of any object.
- **Protected objects** are allowed to use constrained operations with ASER type of request and constrained operations of other protected objects that have no functional activation constraints.
- **Cyclic** and **sporadic objects** can only use constrained operations of non-terminal active objects. They can use constrained operations of protected objects and of other cyclic and sporadic operations.

3.2.4.4.2.2 Include Relationship

The INCLUDE relationships allowed in HRT-HOOD are as follows.

- **Passive objects** are only allowed to *include* other passive objects.
- **Active objects** are allowed to *include* any objects.
- **Protected objects** are allowed to *include* passive objects and other protected objects.
- **Cyclic objects** are allowed to *include* at least one cyclic object with one or more passive, protected, and sporadic objects.
- **Sporadic objects** are allowed to *include* at least one sporadic object with one or more passive and protected objects.

3.2.4.4.2.3 Implemented_By Link

HRT-HOOD provides guidelines for decomposition of operations. Operation decomposition is represented by the *Implemented_By* relationship.

- ASER operations can de decomposed to ASER, ASATC, PSER, or PAER.
- ASER operations with functional activation constraints can be decomposed to operations with functional activation constraints (ASER, ASATC), PSER, or PAER.
- LSER operations can be decomposed to LSER, PSER (with or without functional activation constraints), or LSATC. LSER operations with functional activation constraints can be decomposed to LSER, PSER, or LSATC operations with functional activation constraints.
- HSER operations can be decomposed to HSER, PSER (with or without functional activation constraints), or HSATC operations. HSER operation having functional activation constraints can be implemented by HSER, PSER, or HSATC child operations with functional activation constraints.
- PSER (with or without functional activation constraints) and PAER can only be implemented by operations having the same type of request as the parent.

3.2.4.4.2.4 Data Flow

Data flows are shown in HOOD diagrams with the notation **o->** and a label naming the data. Data flow can be uni-directional or bi-directional. Only major data flows are included in the diagrams to represent information exchange between objects.

3.2.4.4.2.5 Exception Flow

An exception is an error condition that is raised when a program terminates or halts abnormally. An exception flow is shown in HOOD diagrams with the notation **-|->** and a label over it naming the exception. The direction of flow is opposite to the data flow direction. Exceptions are raised in operations and may be propagated and handled by some other operation which is related to it by a *use* relationship. The exception has to be specified in the ODS (section 3.1.1.1.4) of the used object and the using object.

3.2.5 Case Study: Mine Control System

3.2.5.1 System Specification

The ESA developed HRT-HOOD method has been applied to the development of several real-time systems including design of a mine control system and an orbital control system. This paper briefly illustrates the mine control system case study as described in [BW95].

A system is to be designed for operation of a pump control system used for mining. The system is illustrated in Figure 13. The basic task of the system is to pump the water that accumulates at the bottom of the shaft to the surface.

3.2.5.1.1 Functional Requirements

- Pump operation

The pump is used to drain out water when the level reached the high water level and it continues pumping until the water level reached the low level detector. The water flow in the pipe can be detected by means of sensors. The pump can also be operated according to the operator's commands but at all times the operation is possible only if the methane level in the mine is below a critical level.

- Environment monitoring

It is required to monitor the environment for methane levels; the operation of the pump relies on this level being non-critical. In addition, the air flow and carbon monoxide levels are also monitored. If any of these levels violate the norms, alarms should be signaled.

- Operator interaction

An operator controls the system via a terminal and is informed of all critical situations.

- System monitoring

The system is monitored at all times and all events are logged so that they can be checked or analyzed whenever necessary.

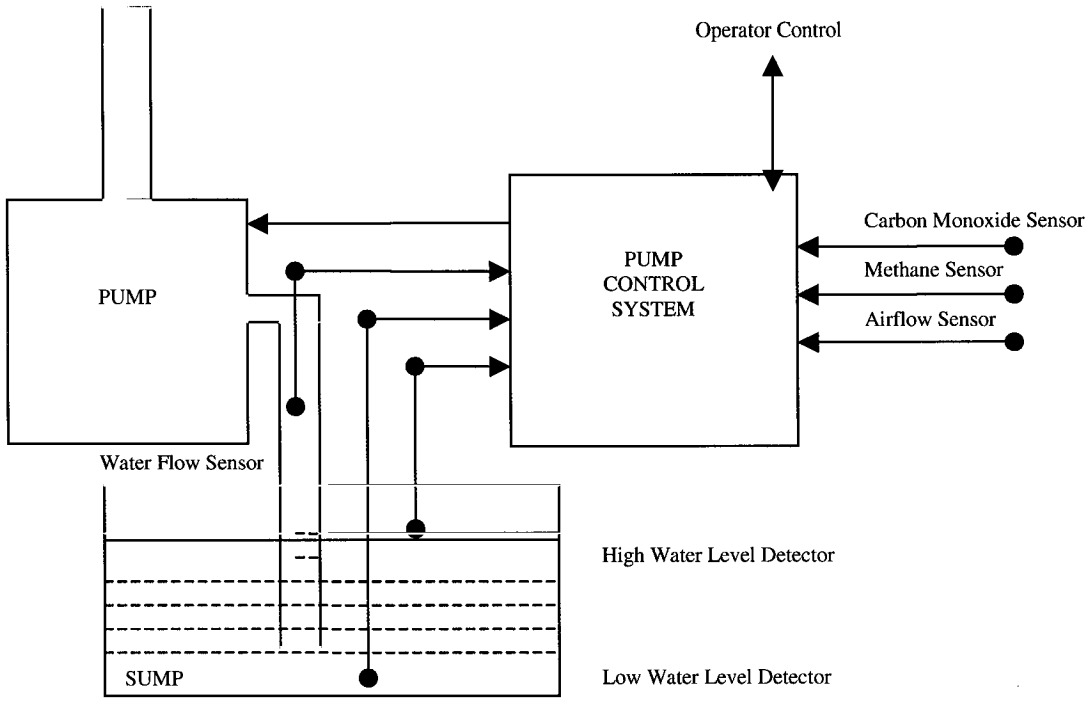

Figure 13. Mine control system

3.2.5.1.2 Non-functional Requirements

Three non-functional requirements are of concern in the mining system – timing, dependability, and security. The case study discusses timing issues, since HRT-HOOD is mainly concerned with handling timing requirements during design.

- Monitoring periods

The maximum period for reading the sensors can be specified. Also, the sensors for detecting water level are event driven and are expected to respond within an acceptable period.

- Shut-down deadline

It is required that the mine be shutdown when the methane level in the mine exceeds the acceptable level. The deadline (D) is related to the rate at which methane accumulates (R), the sampling period (P), and the safety margin between the critical level and the level which causes an explosion (M). The relationship can be specified as $R(P+D) < M$.

- Operator information deadline

Deadlines need to set by which the operator should be informed of events like critical level of methane or other gases, or when the pump fails to operate.

The attributes and their values related to the timing requirements in this case study are illustrated in Table 3.

	Periodic/Sporadic	Arrival Times	Deadline
Methane sensor	P	5.0	1
Carbon Monoxide sensor	P	60.0	1
Water flow sensor	P	60.0	3
Airflow sensor	P	60.0	2
High/Low water interrupt handler (HLW handler)	S	100.00	20

Table 3. Attributes of periodic and sporadic processes [BW94]

3.2.5.2 The Logical Architecture Design of the Mining System

3.2.5.2.1 Top-Level Design

The logical architecture design begins with identifying the top-level components (classes) of the system. From the functional requirements, it can be derived that the classes include the *pump controller subsystem*, the *environment monitor subsystem*, the *operator console subsystem*, and the *data logger subsystem*. Figure 14 describes the relationship among these subsystems and the operations used for interfacing. The purpose of each operation is defined below.

Pump controller

- *not safe* is called by the environment monitor to indicate to the pump controller that it is not safe to operate the pump.
- *is safe* is called by the environment monitor to indicate to the pump controller that it is safe to operate the pump.
- *request status* is called by the operator console to enable viewing of the current system status.
- *set pump* is used by the operator to send commands to the pump system.

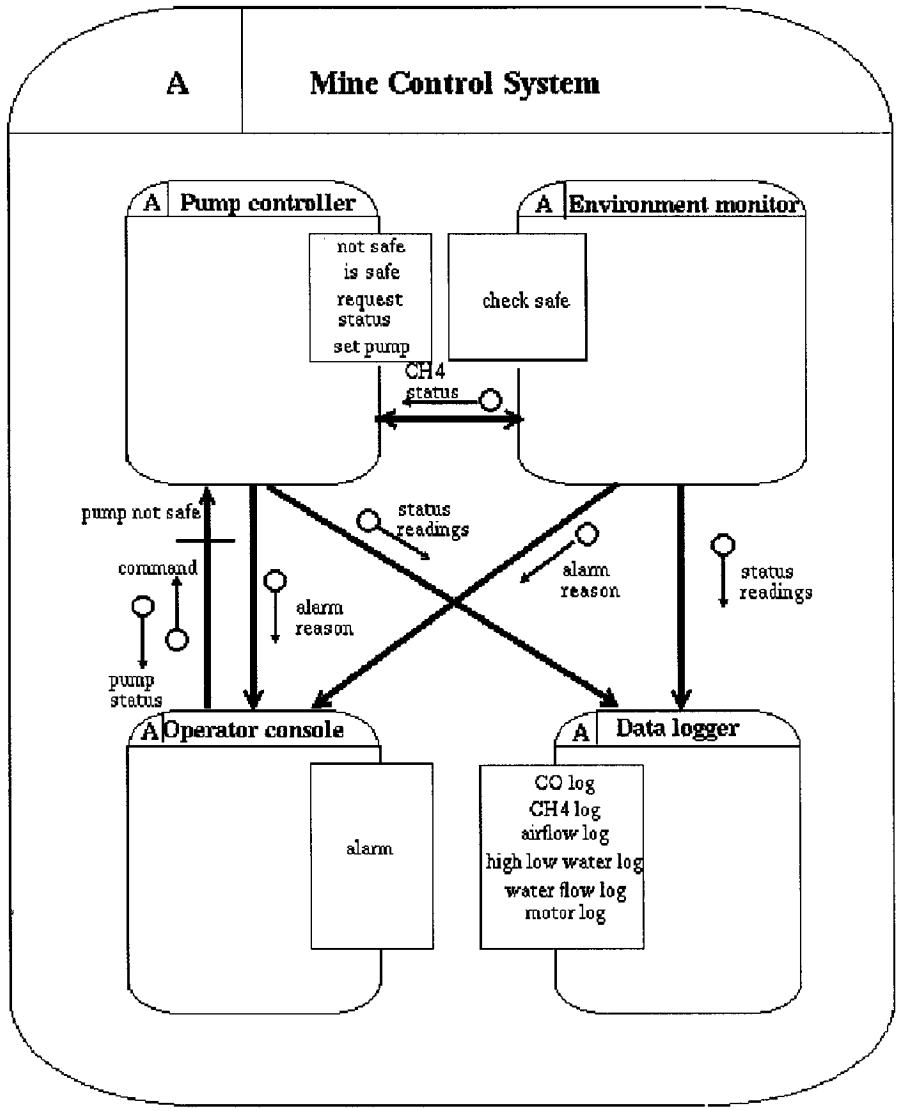

Figure 14. First level hierarchical decomposition of control system [BW94]

Environment monitor

- *check safe* is called by the pump controller to check whether the methane level is below the critical before starting pump operation.

Operator console

- *alarm* is called by the pump controller or the environment monitor if the gas levels exceed the normal.

Data logger

- Six operations are provided in the data logger class and are called by the pump controller and the environment monitor to enable storing of data.

The subsequent figures represents objects via the HOOD notation, operations via rectangles, exceptions by -|-, and data flows by O->.

3.2.5.2.2 Pump Controller

The pump controller subsystem is illustrated in Figure 15. The $\mathbf{Z}_{>}$ is the notation for constrained operations with the constraint specified above the symbol. The operations of the pump controller class are internally implemented by the motor subsystem, which is shown as a protected object. The motor subsystem is also responsible for interfacing with the other top-level systems, namely the environment monitor, operator console, and data logger. The operation decomposition adheres to the rules defined in the HRT-HOOD design method, which is described in Section 3.2.4.4.2.3.

The water flow sensor subsystem is shown as a cyclic object and continuously monitors the water flow in the mine and stores the data in the data logger.

The High low water sensor subsystem is used to handle interrupts from the related sensors and is further decomposed and shown in Figure 16.

3.2.5.2.3 Environment Monitor

The environment monitor subsystem, which is represented as an active object is further decomposed into three cyclic objects, each of which represent the sensors used to monitor the gas levels and one protected object which is used to access data regarding the methane level. The HSER *check safe* operation of the environment monitor is internally implemented by a PSER operation of the CH4 status object. All the objects in the environment monitor are terminal objects. The environment monitor subsystem is shown in Figure 17.

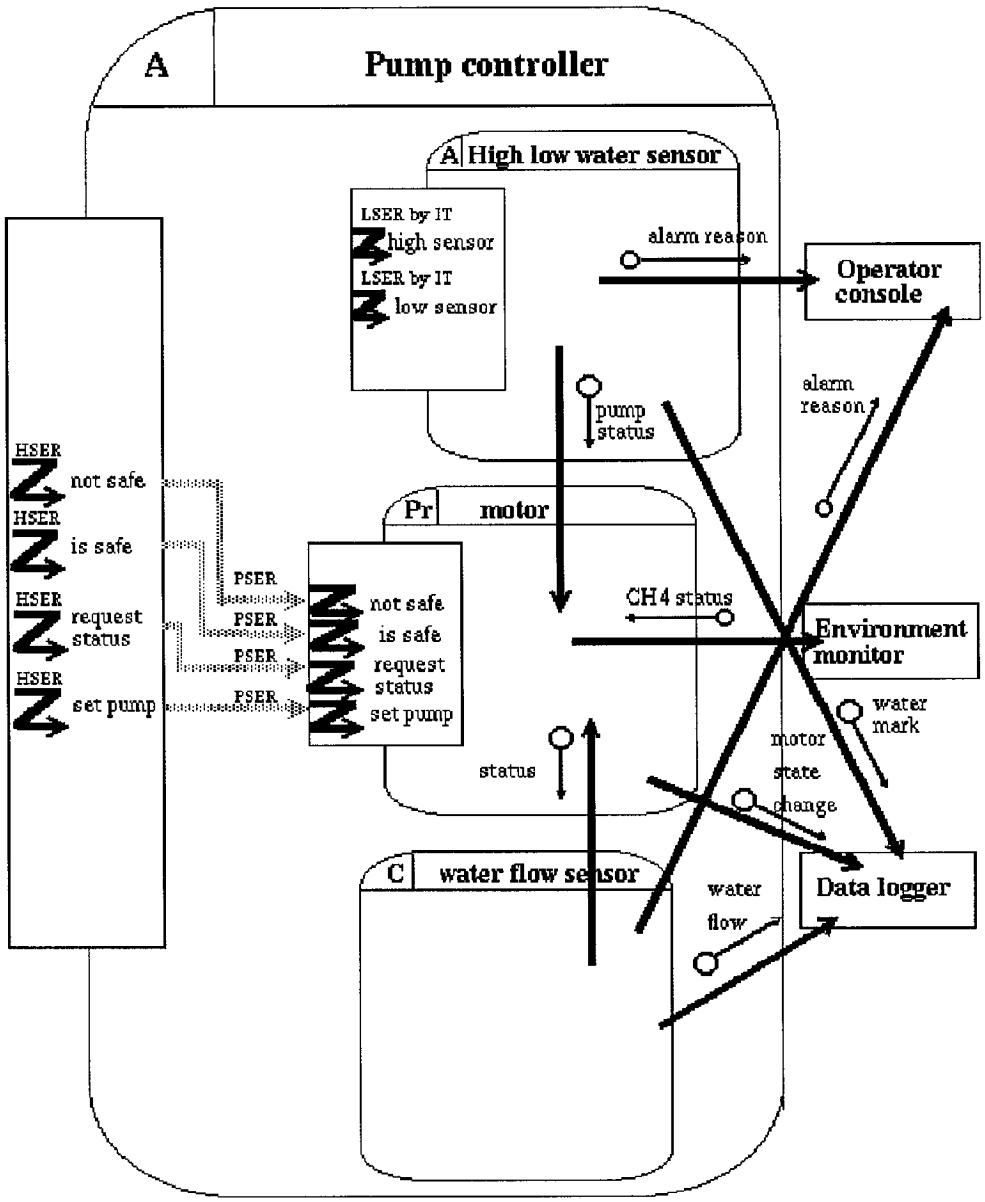

Figure 15. Hierarchical decompositions of the pump object [BW94]

3.2.5.2.4 Operator Console and Data Logger

The operator console and the data logger subsystems are not described here since they are not of much relevance to the HRT-HOOD design objectives. It is however required that the objects provide asynchronous interfaces.

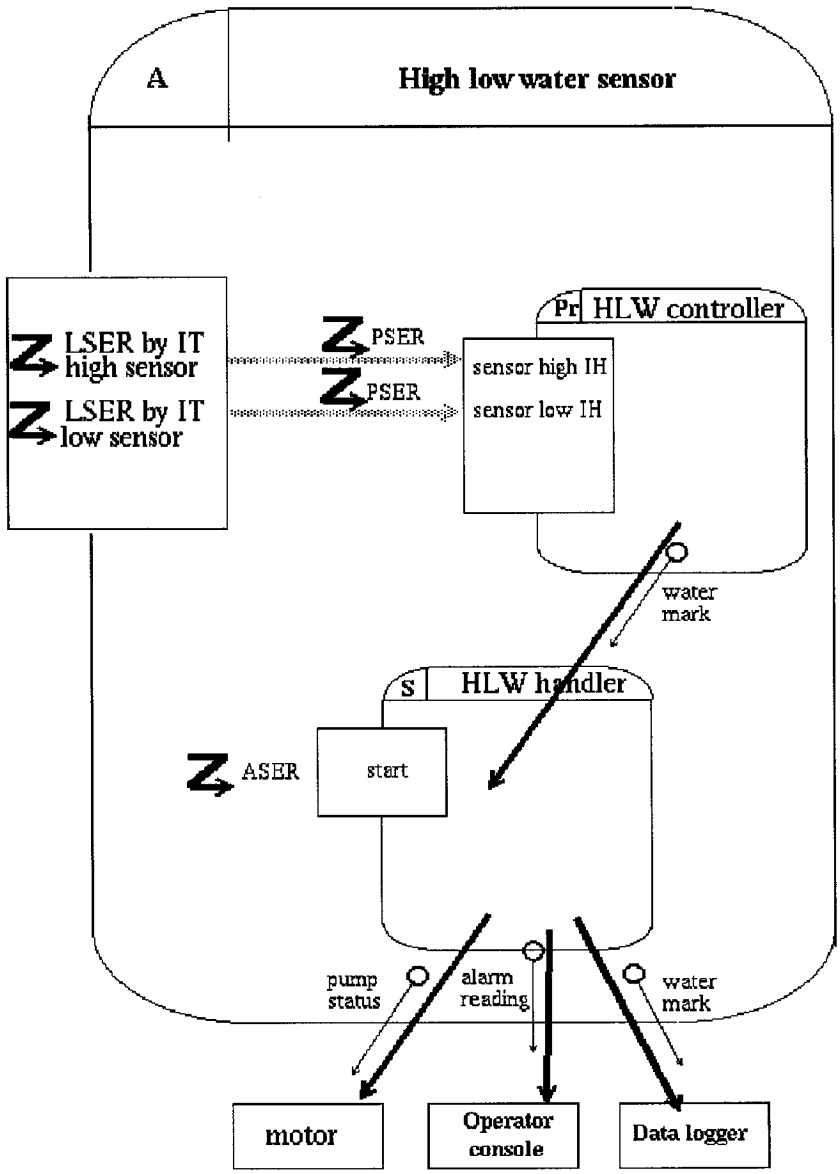

Figure 16. Decomposition of the high low water sensor [BW94]

3.2.5.3 The Physical Architecture Design of the Mining System

This level of design takes into consideration the non-functional requirements of the mining system and assigns timing attributes to objects. The *periodicity* and *deadlines* associated to the periodic and sporadic objects in the system have been defined earlier in Table 3. The HRT-HOOD method also provides a framework that enables schedulability analysis of the terminal objects.

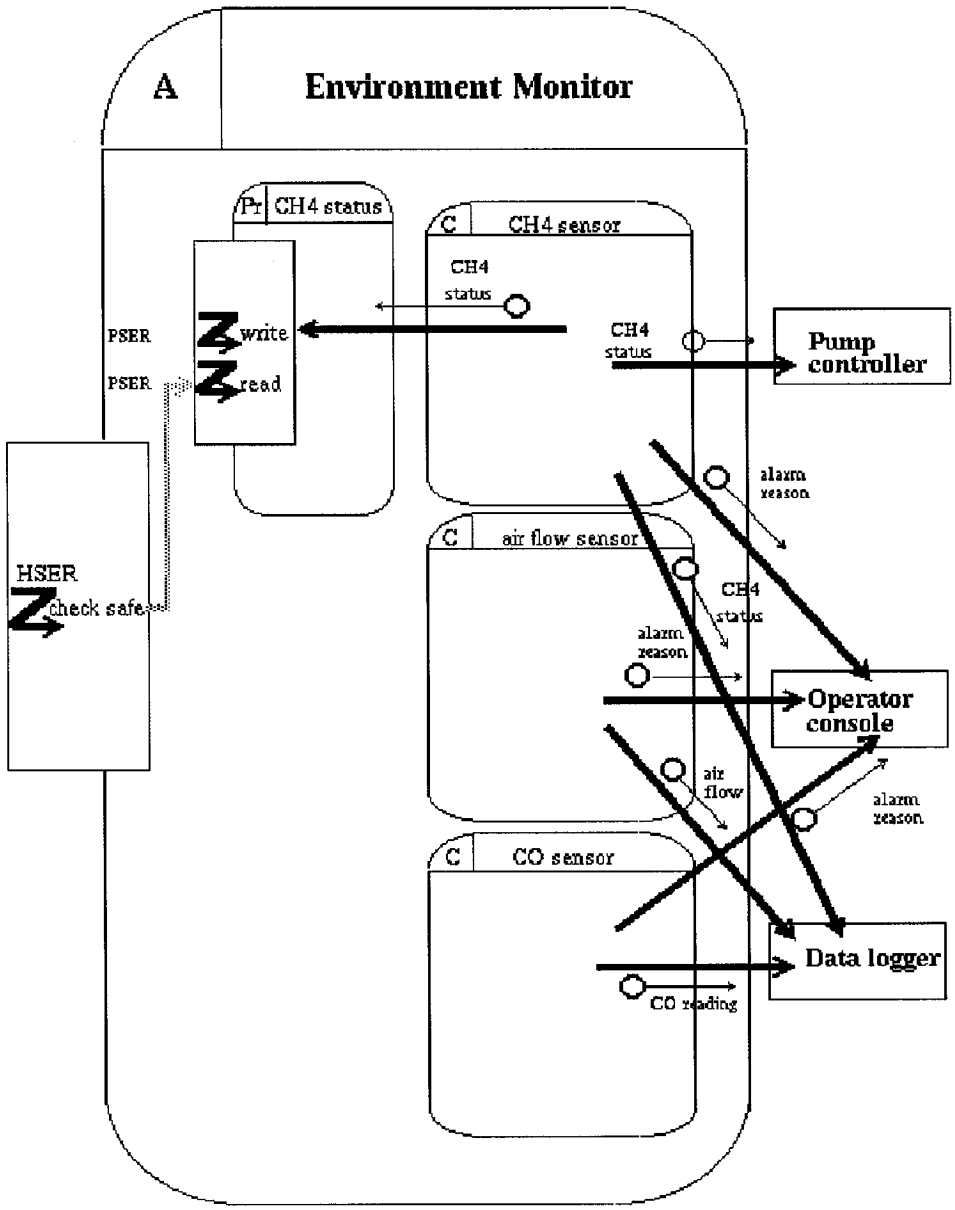

Figure 17. Hierarchical decomposition of the environment monitor [BW94]

3.2.6 Mapping Design to Implementation

HRT-HOOD provides guidelines to map the design to implementation. Details about the mapping can be found in [BW95]. The general concept behind the mapping is as follows:

Each object is mapped to a module/package, and each of the operations is translated to a procedure/function. In addition, active objects include a set of internal tasks and a synchronization agent. Protected objects are mapped to a package and a synchronization

agent. Besides the package, cyclic and sporadic objects include a synchronization agent and a periodic task/process.

The guidelines are specifically intended to convert the design to Ada code. There is an intermediary step in which the design is mapped to an *Object Description Skeleton (ODS)*. A ODS is a structured text format which provides more details than the HRT-HOOD diagrams and is closer to the code. An ODS is required for every object identified in the system. More about the ODS can be found in [BW95].

This thesis is not concerned with implementation details and hence further explanations are not provided here.

3.2.7 Developments of HRT-HOOD

3.2.7.1 Fault Tolerant HRT-HOOD

HRT-HOOD at present does not consider fault-tolerance issues during design. The Real-Time Systems Group at the University of York had initiated a research project *Fault Tolerant HRT-HOOD* in 1997. The goal of the project was to develop an extension of HRT-HOOD which allowed fault tolerant designs to be produced and analyzed [FTHH]. But, the project was discontinued and no concrete work has been done in this area yet.

3.2.7.2 OOHARTS

The Object-Oriented Hard Real Time System (OOHARTS) approach [KN99] is based on HRT-HOOD and UML and proposes a process targeted to develop dependable hard real-time systems. OOHARTS is discussed briefly in Chapter 5, Section 5.1.1.

3.3 Summary

HRT-HOOD should be considered as a new method and not an updated HOOD method. The differences between HOOD and HRT-HOOD are as follows.

- HRT-HOOD has additional object types: *cyclic*, *sporadic*, and *periodic*. The *include* and *use* relationships have been updated to consider these objects.

- Attributes of objects associated to timing requirements have been added to objects in HRT-HOOD. These include deadline, period, worst-case execution time, etc.
- The HRT-HOOD method provides support for analyzing non-functional requirements.

HRT-HOOD is one of the first methods to consider real-time issues during the early software development stages in such detail. The method aims to provide designers with detailed guidelines to develop a system that satisfies the timing requirements. Among the many non-functional requirements, the HRT-HOOD method considers the dependability attributes which are essential to real-time systems: availability, reliability (to some extent) and safety, and also the associated requirement timeliness. The method however does not provide fault-tolerance support, and hence it is up to the designer to incorporate means into the system to achieve dependability in terms of maintainability and security. The method does not suggest ways of identifying the mentioned non-functional requirements, but focuses on how to integrate them into the design phase. It also aids in the transition from design to code generation.

The TARDIS framework described in Chapter 4 uses concepts similar to HRT-HOOD, but also considers ways of satisfying other dependability attributes that are not supported by HRT-HOOD.

Chapter 4.

Fault Tolerance Frameworks and Middleware

4.1 TIRAN

TaIlorable fault toleRANce frameworks for embedded applications (TIRAN) is a European Strategic Programme for Research in Information Technology (ESPRIT) project completed in October 2000. The primary goal of the project was to develop a software framework to provide fault tolerant capabilities to automation systems which would reduce development costs significantly. The framework aims to solve problems in fault-affected applications by considering error detection, isolation and recovery, reconfiguration and graceful degradation. TIRAN is targeted to system manufacturers, system integrators, and software system designers. The candidate application fields are energy and transportation embedded automation systems.

The requirements for such a framework were taken from users and producers of automation systems and generalized so as to fulfill needs of this industrial environment and the external market. The partners of the TIRAN Project are ENEL-R&D (Italy), SIEMENS (Germany), TXT Informatica (Italy), EONIC Systems (Belgium), Katholic University of Leuven (Belgium) and University of Turin (Italy). The framework was developed and experimented on a pilot application from the ENEL plant automation field. ENEL S.p.A. is the main Italian electricity supplier (the third-largest in the world). The pilot application is an ENEL system, called the Primary Substation Automation System (PSAS), consisting of different modules managing an electric substation. The application is a good representation of most dependability requirements of the energy field, which include integrity, security, availability, and EMI (electromagnetic interference) immunity.

TIRAN provides the users of the framework with a methodology for collecting, specifying, and validating FT requirements, with a characterization of framework elements, and guidelines for using the framework. The specification of fault tolerance is based on the Unified Modeling Language, which is a standard graphical modeling method, and TRIO (Tempo Reale ImplicitO) temporal logic, which has been developed by ENEL specifically for real-time systems.

This part of the chapter is mainly based on the information reported in [TIRAN D1.1] and [TIRAN D3.3]. Section 4.1.1 discusses the requirements of the fault tolerance framework. Section 4.1.2 defines the elements of the framework. Section 4.1.3 presents the architecture of the framework. The user support provided by the TIRAN framework is described in Section 4.1.4, which concentrates on the FT specification methodology support. Most of the fault tolerance terms used in this chapter, e.g. basic definitions of faults, error, failures and their types, have already been introduced in Chapter 2 of this thesis.

4.1.1 FT Framework Requirements

4.1.1.1 Requirement List

The requirements for the TIRAN framework can be categorized as follows:

- Functional requirements to be considered include fault handling, monitoring and fault injection, configuration, fault assumptions, and fault tolerance scenarios.
- Performance requirements include hard real-time and soft real-time.
- Architectural requirements concern the framework architecture (backbone, layered approach, composition of mechanisms, etc.) and communication (communication model, interaction).
- Portability requirements need to consider constraints on the target platform, the real-time operating system (RTOS), and the developing environment.
- Quality requirements that need to be accounted for include development process, flexibility, testability, correctness, and verification & validation.

4.1.1.2 Fault Tolerance Scenarios

The TIRAN methodology aims to handle fault tolerance scenarios relevant to automation systems. Table 4 presents the considered faults and failures.

Fault	Affects	Failure	Fault Tolerance Means
Permanent, physical	Processor	Permanent omission	H/W redundancy, Dynamic reconfiguration
Temporary, physical	Processor	Temporary or permanent omission	Fault Diagnosis, Temporal Redundancy
Permanent/temporary physical	Memory subsystem	Byzantine	Stable memory / spatial redundancy / exception handler
Permanent/temporary physical	Communication subsystem	Permanent omission	H/W link redundancy, Fault masking
Permanent, external	-	Byzantine	Use replicated components placed in different locations/ Confinement or fault containment
Temporary, external	-	Byzantine	Temporal redundancy tool
Permanent/temporary design	-	Byzantine	Design diversity/Comparison

Table 4. Fault tolerance scenarios handled by the TIRAN methodology

4.1.2 TIRAN Framework Elements

The TIRAN framework is an integrated set of elements providing FT services for real-time and/or distributed systems. It is composed of the following elements.

- A *library* of basic tools implementing fault tolerance mechanisms
- A *backbone* coordinating the basic tools
- A *language* expressing configuration and recovery strategies.

Information about the elements has been taken from [TIRAN D3.3].

4.1.2.1 Library

The framework provides a library of basic tools to support fault tolerance mechanisms including error detection, recovery, and fault masking. "The tools are adaptable, parametric, software-based implementations of the following fault tolerance mechanisms: watchdog (WD), distributed memory (DM), local voter (LV), output delay (OD), stable memory (SM), distributed synchronization (DS) and time-out management system

(TOM)" [TIRAN D3.3]. The tools can be used independently or jointly with other tools. Table 5 shows how these mechanisms relate to the different FT steps.

	WD	DM	LV	OD	SM	DS	TOM
Error detection	✓	✓	✓		✓		
Error recovery		✓	✓		✓		
Fault masking		✓	✓	✓	✓		
Fault containment	✓	✓	✓	✓	✓		
System coordination						✓	✓

Table 5. Association of FT mechanisms to FT steps

4.1.2.2 Backbone

A control backbone is a distributed application that extracts information about the application's topology, its progress and its status; it maintains this information in a replicated database and it coordinates fault tolerance actions at runtime via the interpretation of user-defined ARIEL (described in Section 4.1.2.3) recovery strategies. The backbone functions as a middleware, hierarchically structured to maintain a consistent system view and containing self-testing and self-healing mechanisms.

4.1.2.3 Recovery Language

The TIRAN framework makes use of a language ARIEL to set the properties and parameters of the basic tools and to specify the recovery strategies. An ARIEL script is composed of a declarative part, which is used to configure the tools, and a recovery part, which describes the recovery actions.

4.1.3 Framework Architecture

"The TIRAN framework acts as a middleware below the application level and on top of the system level, so as to allow the system to deliver a required service in spite of error occurrences" [TIRAN D3.3]. Figure 18 illustrates the general architecture structure.

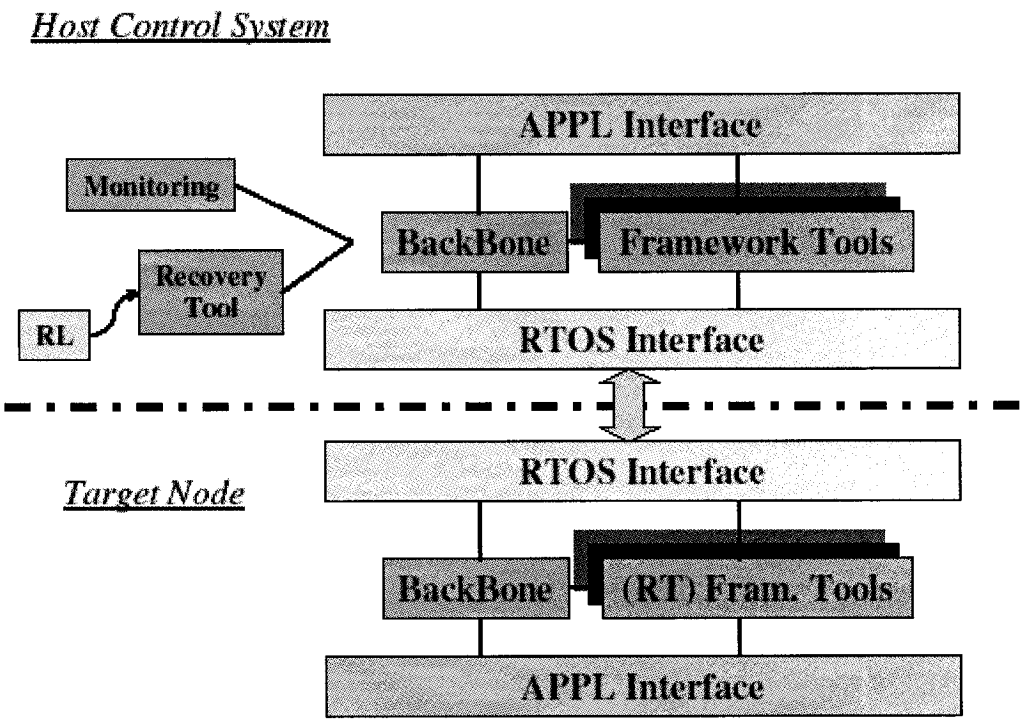

Figure 18. TIRAN framework architecture [TIRAN D3.3]

4.1.4 User Supports

The TIRAN framework addresses error detection, isolation and recovery, reconfiguration and graceful degradation of fault-affected applications.

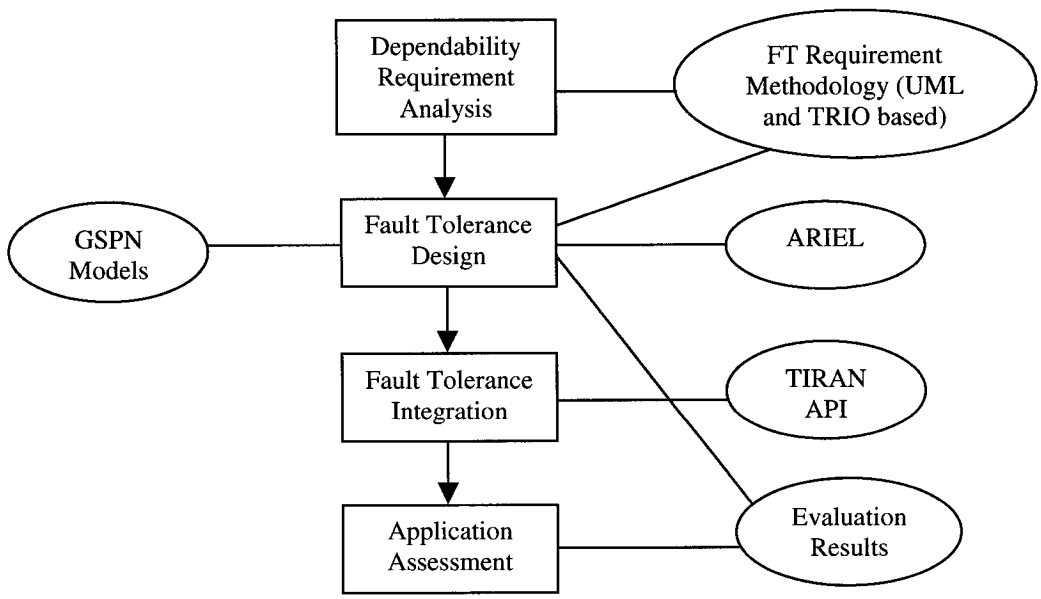

Figure 19. TIRAN framework in the development process [TIRAN D3.3]

The methodology by which TIRAN supports the specification and analysis of fault tolerance requirements is outlined below.

Step 1. Support to FT specification – semi-formal approach provided by UML.

Step 2. Formalization of the FT specification – formal method TRIO used to express the FT requirements.

Step 3. Analysis of formal FT requirements – uses TRIO formal techniques to perform V&V activities.

Step 4. Techniques for identifying an appropriate FT solution to fulfill FT requirements specification.

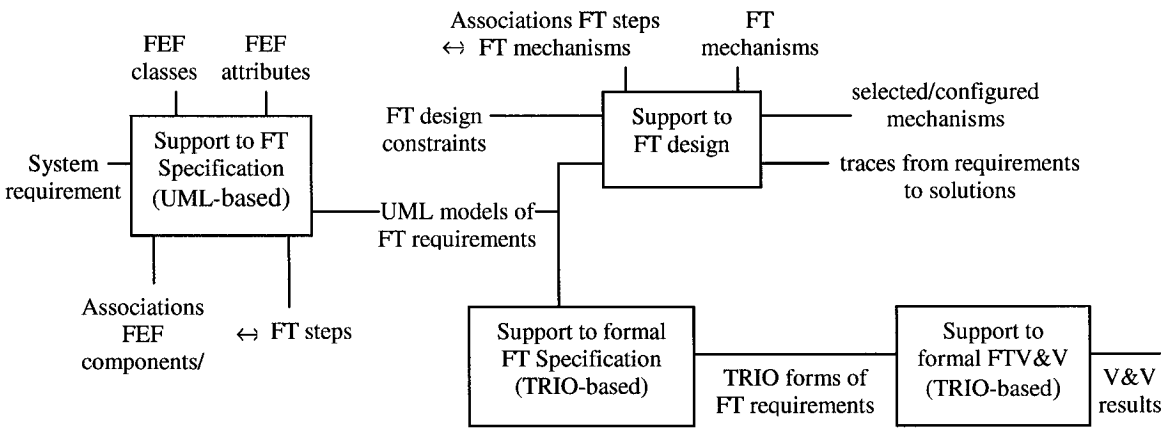

Figure 20. A functional view of the scheme [TIRAN D1.1]

4.1.4.1 Support for FT specification

The first stage in the methodology involves semi-formal specification of FT requirements based on UML. This focuses on

- Fault/Error/Failure (FEF) class hierarchies and attributes
- Associations of FEF classes to system components/functions
- FT step class hierarchy

A meta-model is defined in UML, which is used by applications which follow this methodology for specifying their fault-tolerant requirements. The methodology depends on a set of class hierarchies which are structured as **Packages**[3].

All Packages of the meta-model are described with a generalized class diagram in this section. Specialized versions of the class diagram with respect to the PSAS system are shown in Section 4.1.5.

4.1.4.1.1 Package Methodology

The top-level Package Methodology, as shown in Figure 21, includes child packages System Model, FEF Model, and FT Strategy Model.

- *System Model* addresses system requirements related to FT specifications.
- *FEF Model* supports the description of fault, error, and failure classes, and models the FEF chain.
- *FT Strategy Model* identifies FT steps.

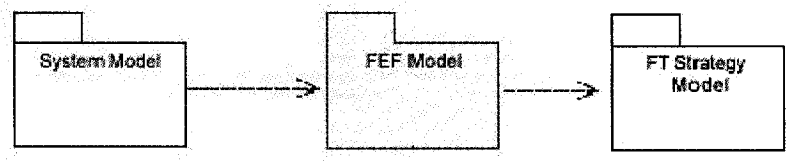

Figure 21. Class diagram of Package Methodology [TIRAN D1.1]

4.1.4.1.2 Package System Model

The Package System Model, as shown in Figure 22, consists of the following child Packages.

- *Package Composition* identifies the structure of the FT system and defines the hierarchies of components.
- *Package Functions* associates the components to its function.
- *Package Dependability Attributes* associates dependability attributes to components and/or functions.
- *Package Time Requirements* associates real-time requirements to components and/or functions.

[3] Package will be capitalized throughout this section to emphasize it as UML terminology.

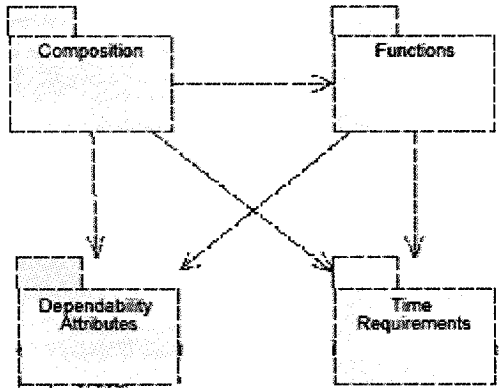

Figure 22. Class diagram of Package System Model [TIRAN D1.1]

4.1.4.1.2.1 Package System Composition

This Package, illustrated in Figure 23, defines the system structure and decomposes system components to create fault-confinement regions with the intention of avoiding fault propagation. UML associations in the diagram show interfaces between classes of components.

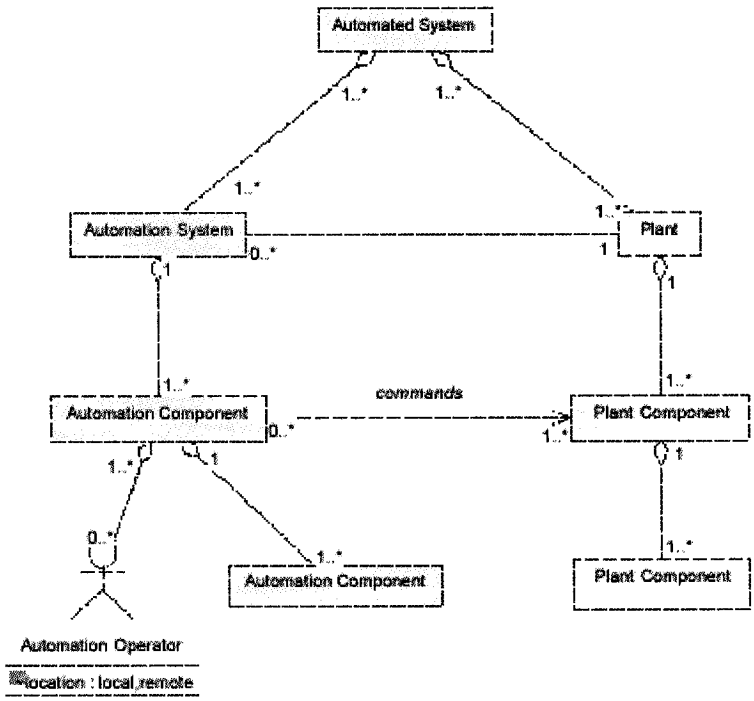

Figure 23. Class diagram of Package System Composition [TIRAN D1.1]

4.1.4.1.2.2 Package System Functions

This Package, illustrated in Figure 24, specifies how the system functions are associated to their components and operators. The attributes of the Communication Function class include *transmission*, *channel*, *bandwidth*, and *location*. The attribute *location* is necessary to know whether a communication is internal or external to a component.

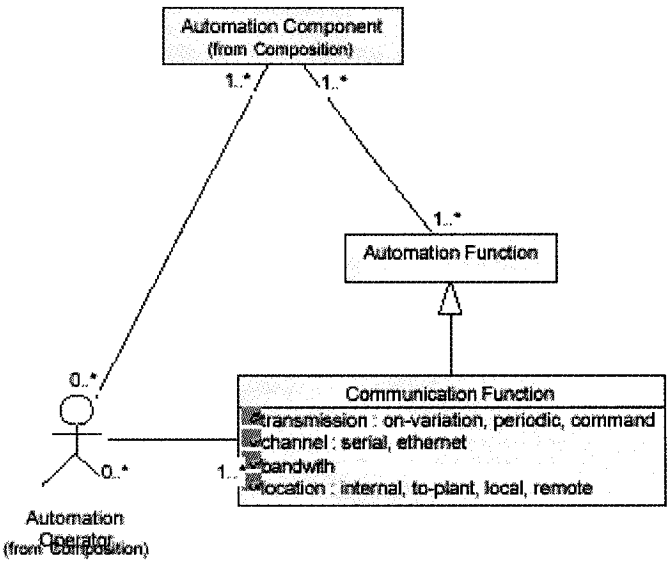

Figure 24. Class diagram of Package System Functions [TIRAN D1.1]

4.1.4.1.2.3 Package Time Requirements

This Package, shown in Figure 25, considers two attributes essential in real-time systems: *cycle time (T_C)*, related to the interaction between the system or component and the plant, and *execution time (TE),* which refers to the maximum response time required by a function. The minimum of execution times among all functions should be less than the cycle time [GO2000].

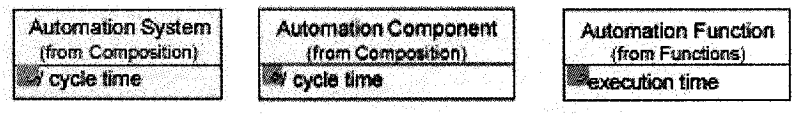

Figure 25. Class diagram of Package Time Requirements [TIRAN D1.1]

4.1.4.1.2.4 Package System Dependability

The scheme entails definition of all relevant dependability attributes in this Package, as shown in Figure 26. The attributes of concern to repairable systems include *criticality*, *complexity*, *MTTF* (*MeanTimeToFailure*), *MTTR* (*MeanTimeToRepair*), *MTBF* (*MeanTimeBetweenFailures*), and *availability*. In the case of non-repairable systems, only the first three attributes need to be considered.

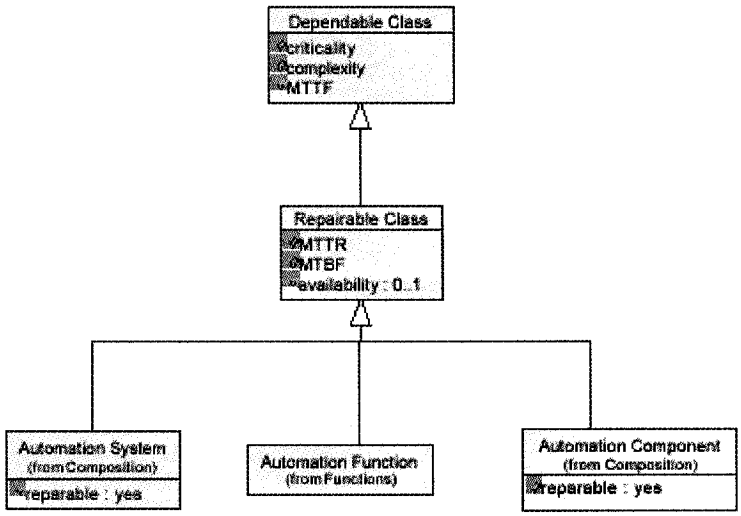

Figure 26. Class diagram of Package System Dependability [TIRAN D1.1]

4.1.4.1.3 Package FEF Model

The FEF Model includes the three participants in the FEF chain, namely, the Fault Model, Error Model, and Failure Model.

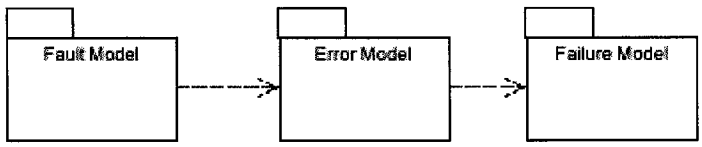

Figure 27. Main class diagram of Package FEF [TIRAN D1.1]

4.1.4.1.3.1 Package Fault Model

The class diagram in Figure 28 illustrates the hierarchies of possible faults and associates a fault to the component where it is located. The Fault Model is derived from the fault

classification scheme defined by Laprie [JL98]. The faults related to automation systems are considered only, and these include physical or design faults. Two types of design faults are considered: systematic faults and intentional faults. Both permanent and temporary physical faults are taken into account, and these are further categorized into development, operational, intermittent, and transient faults. Definitions of these faults can be found in Chapter 2.

Each fault is described by the attributes *fault rate*, *location*, and *latency*. In case of physical faults, the *duration* and *source* of the faults need to be considered as well.

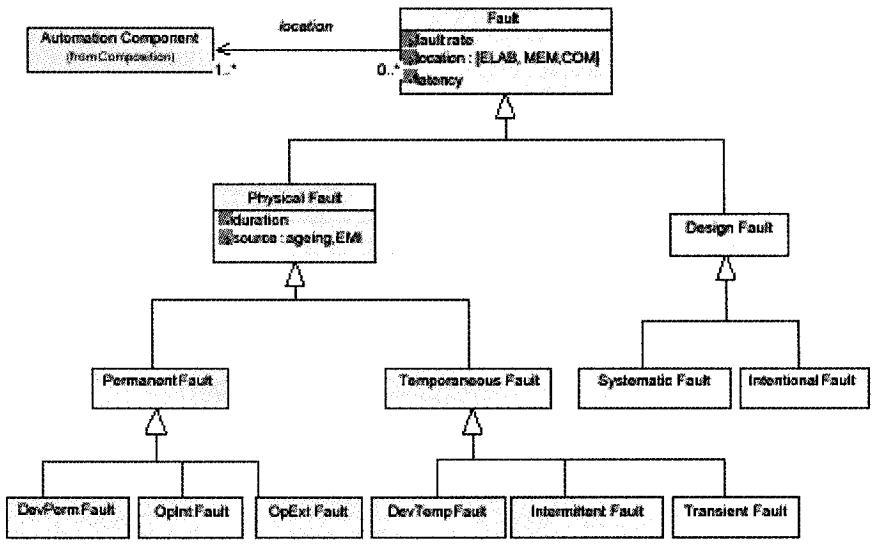

Figure 28. Class diagram of Package Fault Model [TIRAN D1.1]

4.1.4.1.3.2 Package Error Model

The class diagram in Figure 29 illustrates the hierarchy of errors that can occur in the application system. They arise due to faults affecting system components and are related to functions of the faulty component by a multiple association *location*. *Latency* and *probability of the error* are attributes of the root class. Some attributes are added locally to error sub-classes.

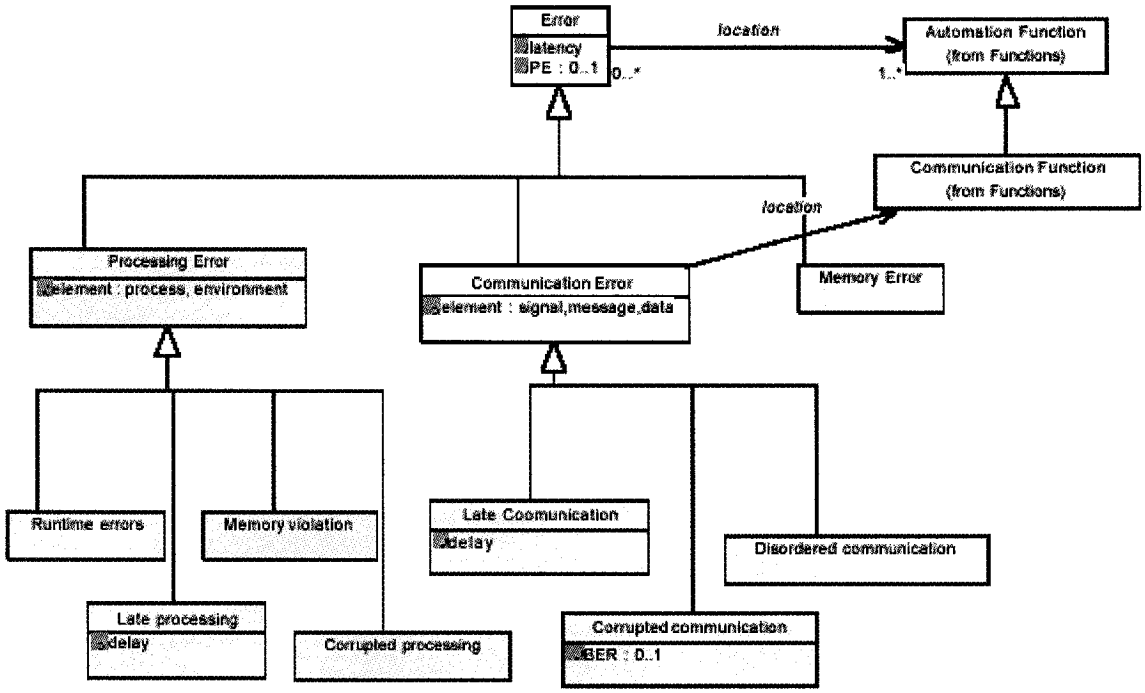

Figure 29. Class diagram of Package Error Model [TIRAN D1.1]

4.1.4.1.3.3 Package Failure Model

The class diagram in Figure 30 illustrates the different modes of failures that can occur in the system. The *criticality* and *probability* of the failure are important attributes of the root class.

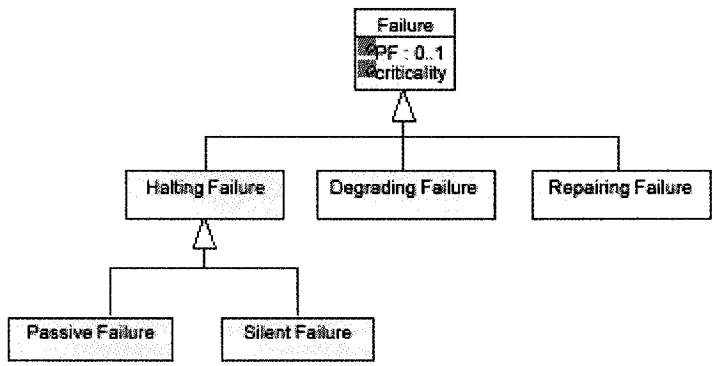

Figure 30. Class diagram of Package Failure Model [TIRAN D1.1]

4.1.4.1.3.4 FEF Chain

The class diagram below illustrates the cause-effect relationships in the fault-error-failure (FEF) chain.

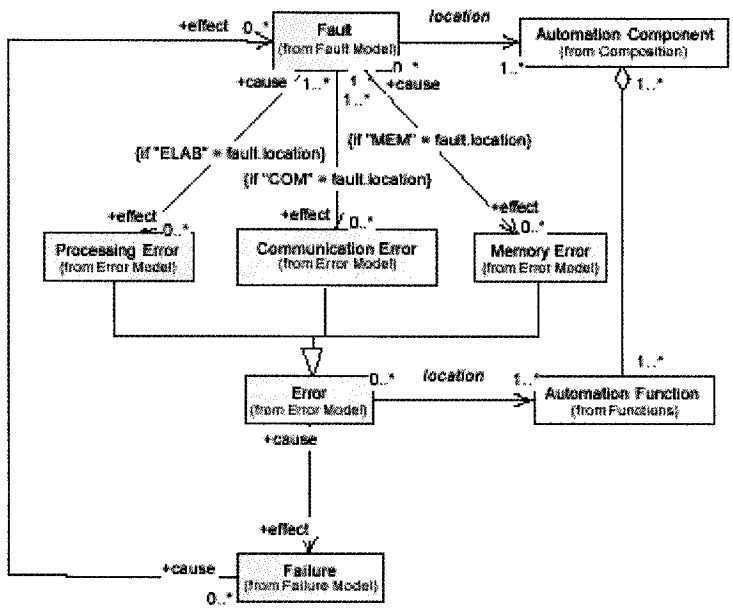

Figure 31. FEF Chain class diagram of Package FEF Model [TIRAN D1.1]

4.1.4.1.4 FT Strategy Model

The Fault Tolerant Strategy Model, shown in Figure 32, classifies the FT steps in three categories following the classification defined by Laprie: Fault Processing, Error Processing, and Fault Treatment.

Fault Processing includes two steps: *fault masking* helps in providing an error-free service and *fault containment* confines the faults in components using spatial or temporal redundancy.

Error Processing involves *error detection*, *diagnosis*, *isolation*, and *recovery* and helps the system regain an error-free state.

Fault Treatment involves *failure handling*, *compensation*, *repair*, *diagnosis*, and *passivation* (system reconfiguration).

The FT steps in this package are connected to the fault and error classes by means of associations (*tolerates*).

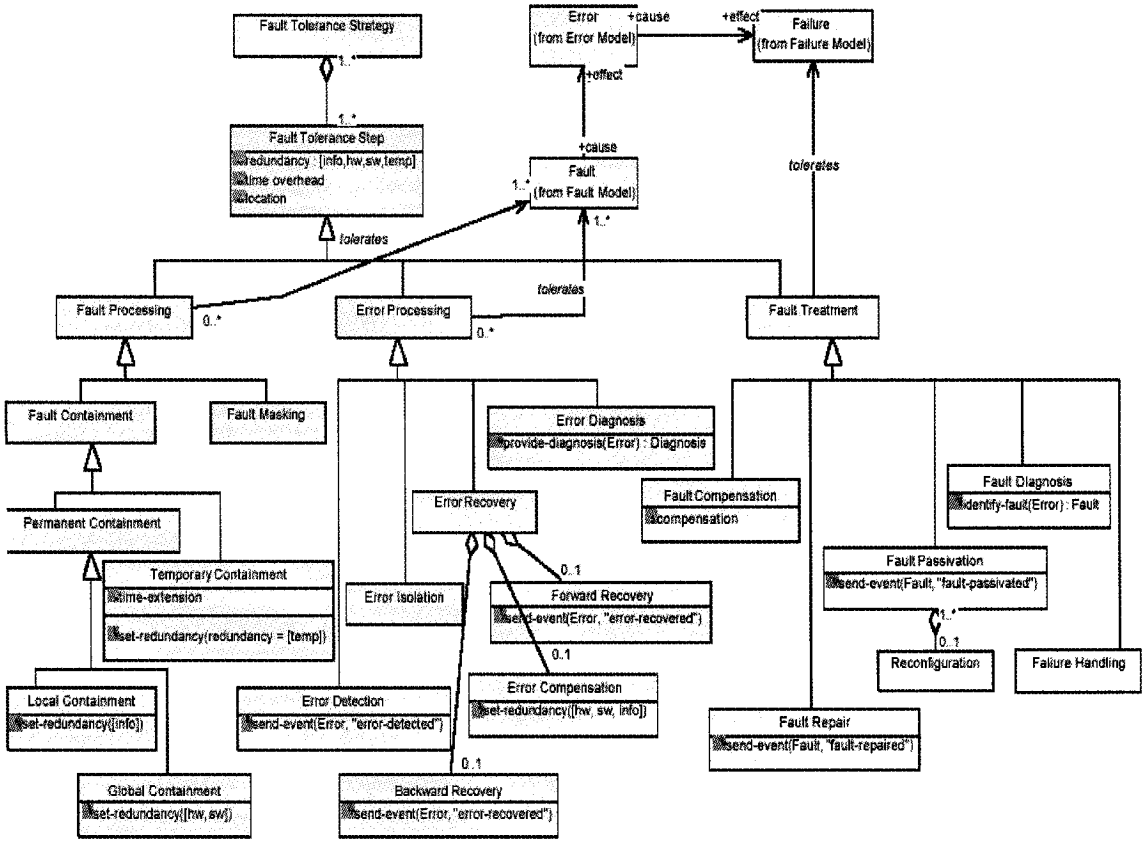

Figure 32. Class diagram of Package FT Strategy Model [TIRAN D1.1]

4.1.4.2 Support to Formal FT Specification

The second stage in the methodology introduces the formal method TRIO, an ENEL product developed for real-time systems. The UML specifications are translated to formal specifications based on TRIO logic. In this step, some parts are added to the formalization for Verification and Validation (V&V) purposes. "Formal methods are typically endowed with automatic analysis capabilities requiring that the formalization be adequately instrumented for performing that type of analysis" [TIRAN D1.1]. More information about the formalization process is available in [TIRAN D1.1].

4.1.4.3 Support to Formal FT V&V

In the third stage, TRIO techniques are applied to the formal FT specification for V&V in two steps: static analysis of the FT strategy and dynamic analysis of system behaviors. The techniques supported by the TRIO method include model generation, history checking, and test case generation. For the PSAS system, some issues that can be analyzed include "which fault classes are associated to which system components?", "which are the FT steps addressing permanent faults?", and "find PSAS histories stabilizing a new state" [TIRAN D1.1]. [TIRAN D1.1] discusses in some details how the V&V analysis is supported in TIRAN.

4.1.4.4 Support to FT Design

In this last stage, the following design activities need to be considered.

- choice of FT mechanisms – the FT mechanisms that can be associated with the specified FT steps need to be selected.
- choice of the target platform
- verification of design constraints – The bounds associated to the mechanisms and platform need to adhere to the time/space/redundancy constraints.

In the case of the PSAS system, the design decisions are influenced by the PSAS FT strategy model and the PSAS time requirements model, which also provide information on FT mechanisms and design constraints.

4.1.5 Case Study: Primary Substation Automation System

The pilot application for the TIRAN project was ENEL's PSAS application. This section illustrates how the TIRAN methodology can be applied to the PSAS application for collecting and specifying FT requirements. Details on this process and further information on validating these requirements can be found at [TIRAN D1.1]

4.1.5.1 FT Requirements

The fault tolerance, real-time, and dependability requirements of the PSAS pilot application reported in [TIRAN D1.1] are listed in

The requirements have been collected following the methodology defined in the TIRAN framework.

4.1.5.1.1 Dependability Attributes

The dependability attributes of concern to the PSAS application include availability, integrity, and security.

- Availability - The availability values are associated to each system component and are assigned based on their criticality level. Availability is quantified on the basis of the MTBF or MTTF.
- Data Integrity – Integrity and security values are evaluated based on the coverage metrics.

4.1.5.1.2 Fault Characterization

The types of faults that need to be considered for this application are permanent, intermittent, and transient faults that affect the system functions and/or physical components.

4.1.5.2 FT Specification of PSAS System

Applying the FT specification scheme, described in Section 4.1.4, to a given system entails the following.

- The sub-parts of the model relevant to the system should be selected.
- The class attributes in the models should be assigned value.
- Sub-parts of models from the scheme should be refined or specialized.

This section demonstrates the results on applying the TIRAN scheme to the PSAS system. The first model that is produced based on the package Methodology (Figure 21) is the Package PSAS. The scheme introduces three inner packages: PSAS System Model, PSAS FEF Model, and PSAS FT Strategy Model. These are further refined in the next sections.

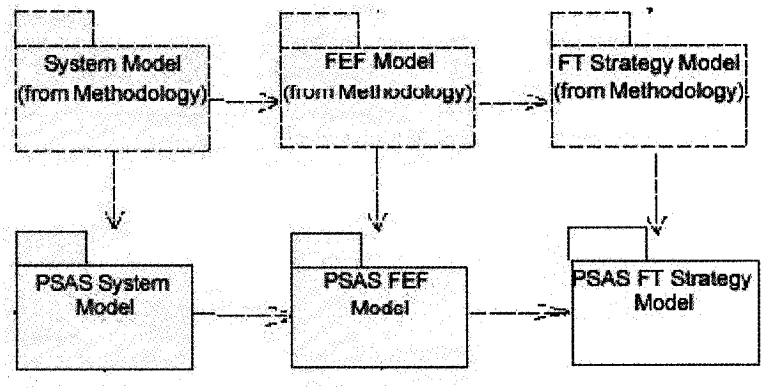

Figure 33. Class diagram of Package PSAS Methodology [TIRAN D1.1]

4.1.5.2.1 Package PSAS System Model

The System Model (Figure 22) is specialized with system-specific classes representing components, functions, time requirements, and dependability attributes. The PSAS System is shown in Figure 34.

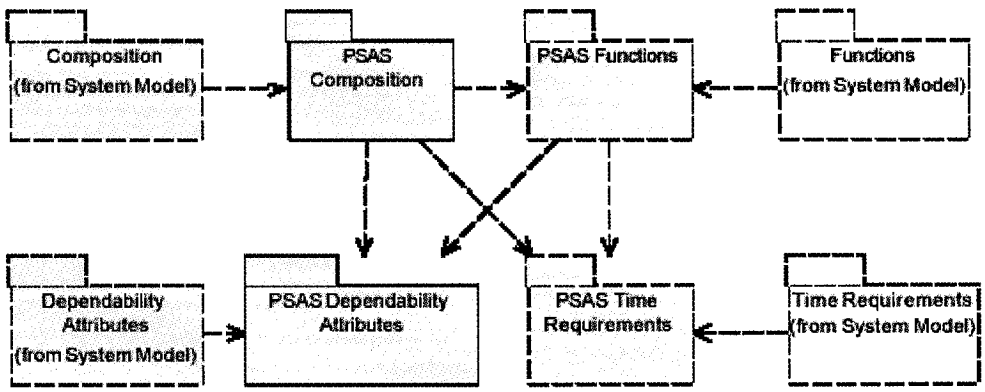

Figure 34. Class diagram of Package PSAS System Model [TIRAN D1.1]

Each of the sub-parts in the model is further refined to give the system-specific models. The PSAS Composition model, shown in Figure 35, is derived from the scheme model in Figure 23 based on the system requirements, SR1 and SR2 defined in Appendix B. SR2 also has specifications for PSAS System Functions model shown in Figure 36, which has been refined from Figure 24. The timing requirement TR1 is modeled and shown in Figure 37 as the PSAS Time Requirements model and is a refinement of Figure 25. The

PSAS Dependability Attributes model, shown in Figure 38, is derived from the dependability requirement DR1, and is based on the model in Figure 25.

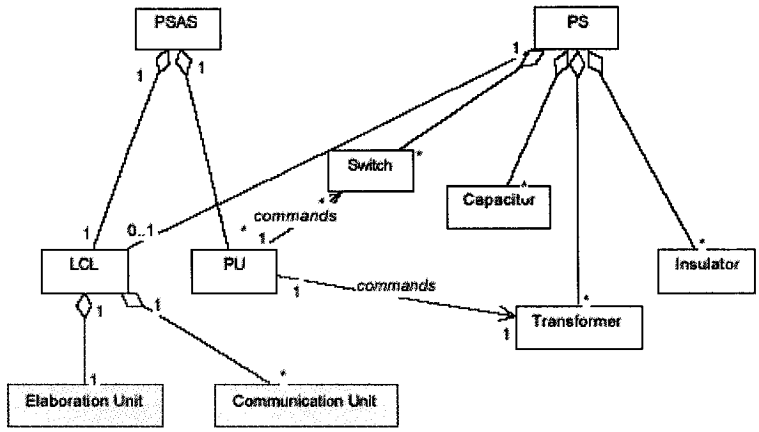

Figure 35. Class diagram of Package PSAS System Composition [TIRAN D1.1]

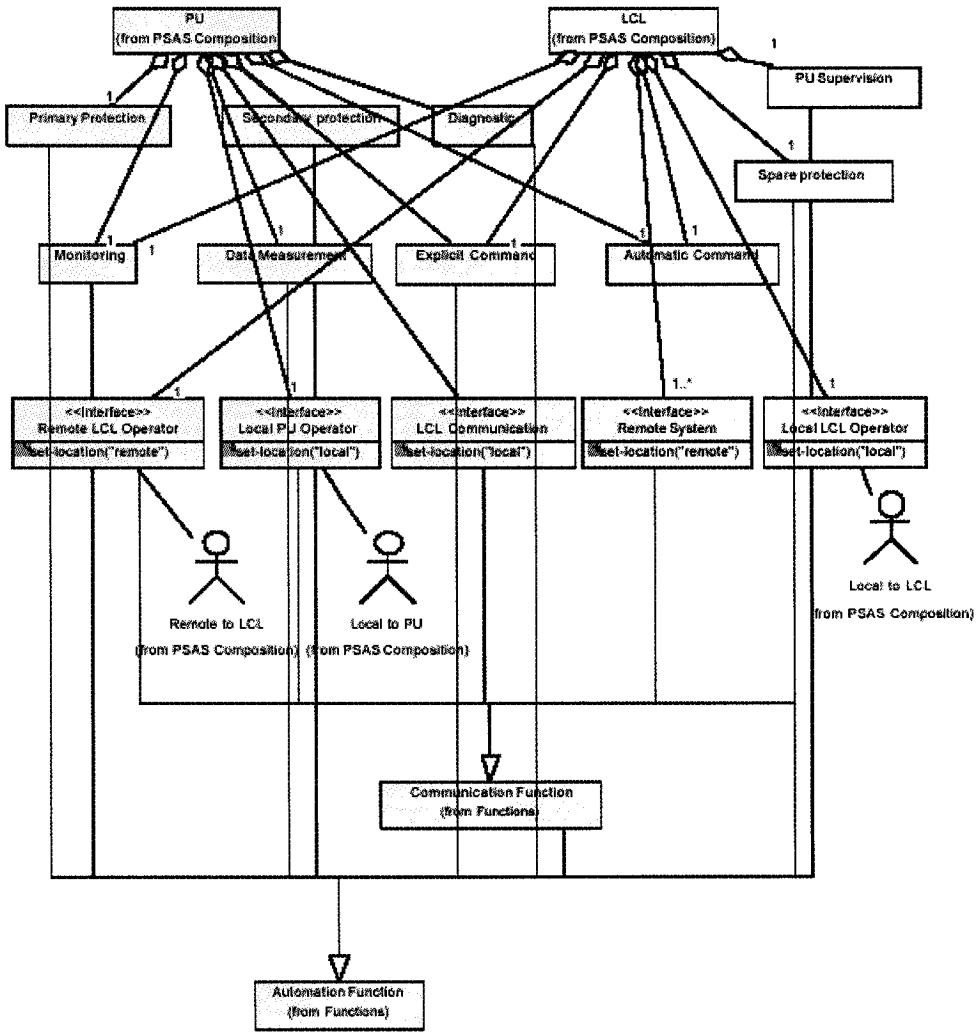

Figure 36. Class diagram of Package System Functions [TIRAN D1.1]

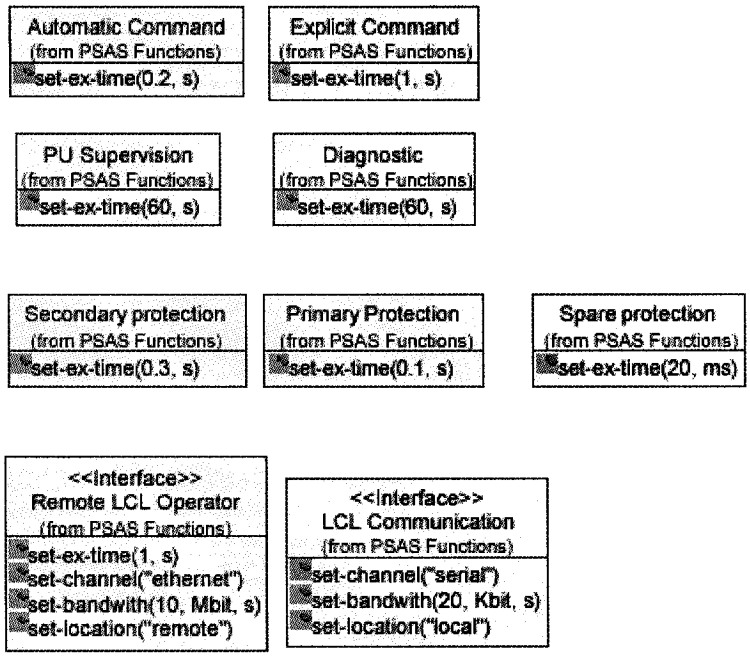

Figure 37. Class diagram of Package Time Requirements [TIRAN D1.1]

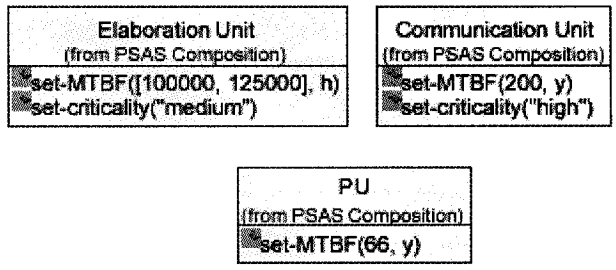

Figure 38. Class diagram of Package Dependability [TIRAN D1.1]

4.1.5.2.2 Package FEF Model

The Fault Model in Figure 28 is customized to give the PSAS Fault Model shown in Figure 39. The customization is based on the FT requirement FR1. FR2 and FR5 concern PSAS errors, which are modeled and shown in Figure 40. Requirements FR3 and FR4 describes the PSAS failures and the respective model is shown in Figure 41.

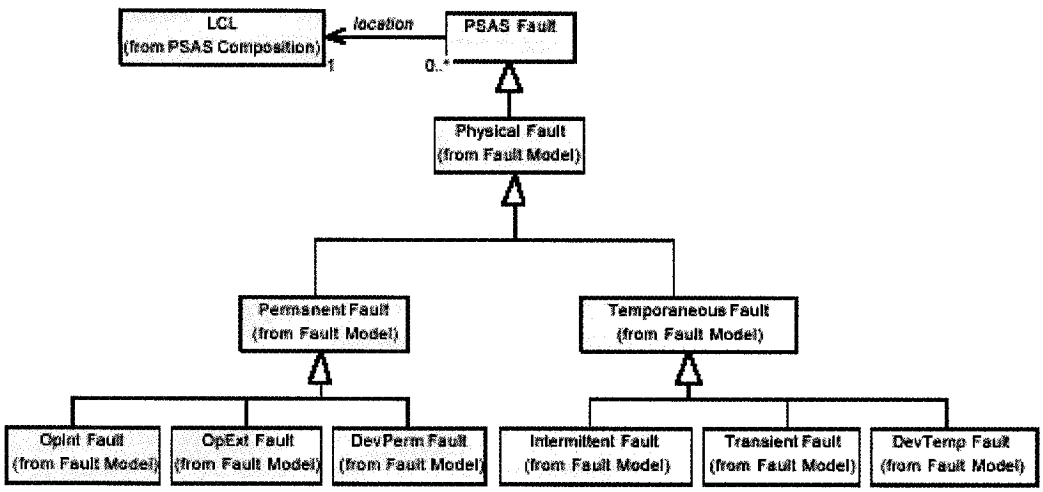

Figure 39. Class diagram of Package PSAS Fault [TIRAN D1.1]

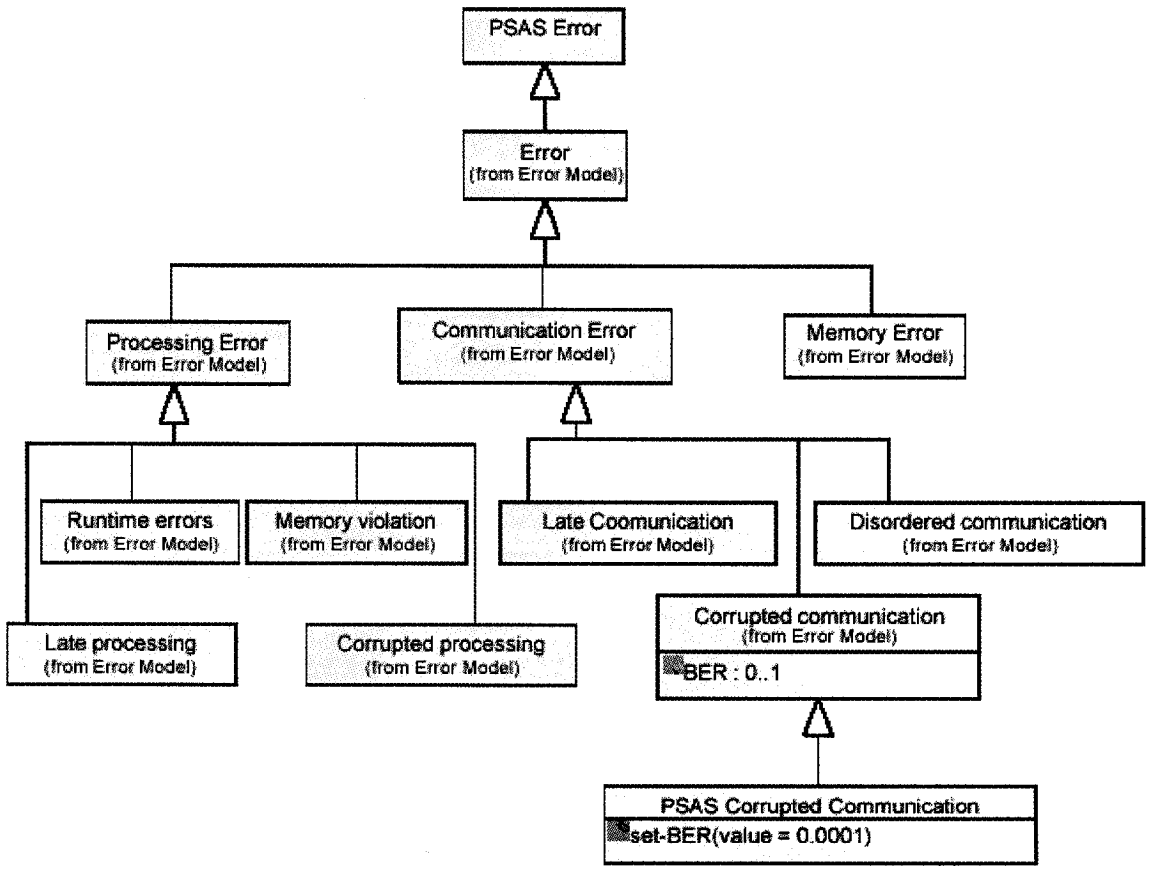

Figure 40. Class diagram of Package PSAS Error [TIRAN D1.1]

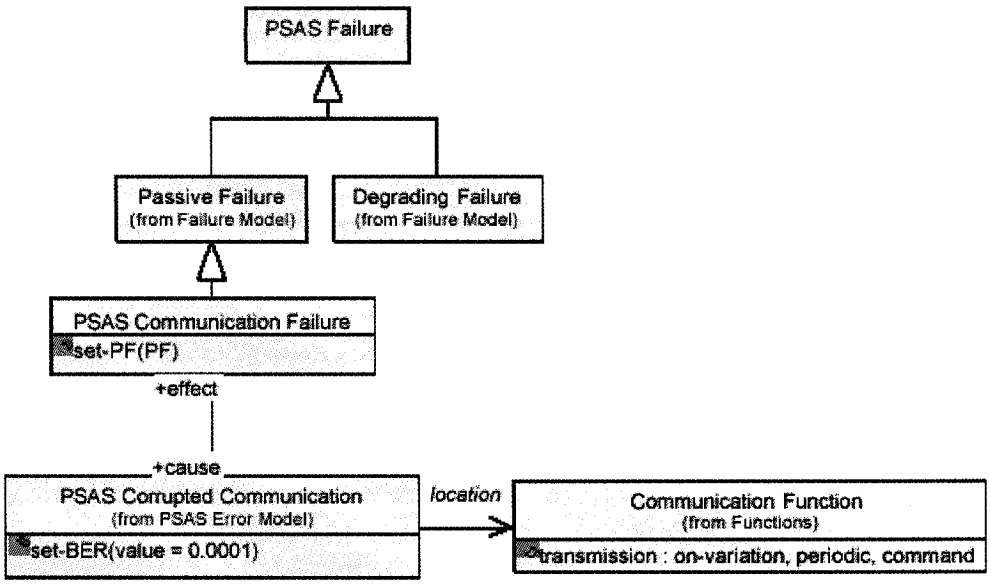

Figure 41. Class diagram of Package PSAS Failure [TIRAN D1.1]

4.1.5.2.3 Package FT Strategy

The PSAS FT Strategy is expressed by PSAS requirements FR2-FR19 and is shown in Figure 42. Finally, Figure 43, Figure 44, and Figure 45 illustrate the association between the PSAS FT steps and the fault/error/failure types in the PSAS fault/error models.

4.1.6 Real-time Applications

The TIRAN framework does provide support for real-time properties but the communication mechanisms are not suited to applications which require "ultra-low response times" [TIRAN D3.3]. The framework was designed with the portability goal in mind, and hence an overhead is incurred due to the tuning of the communication mechanisms to the target platform.

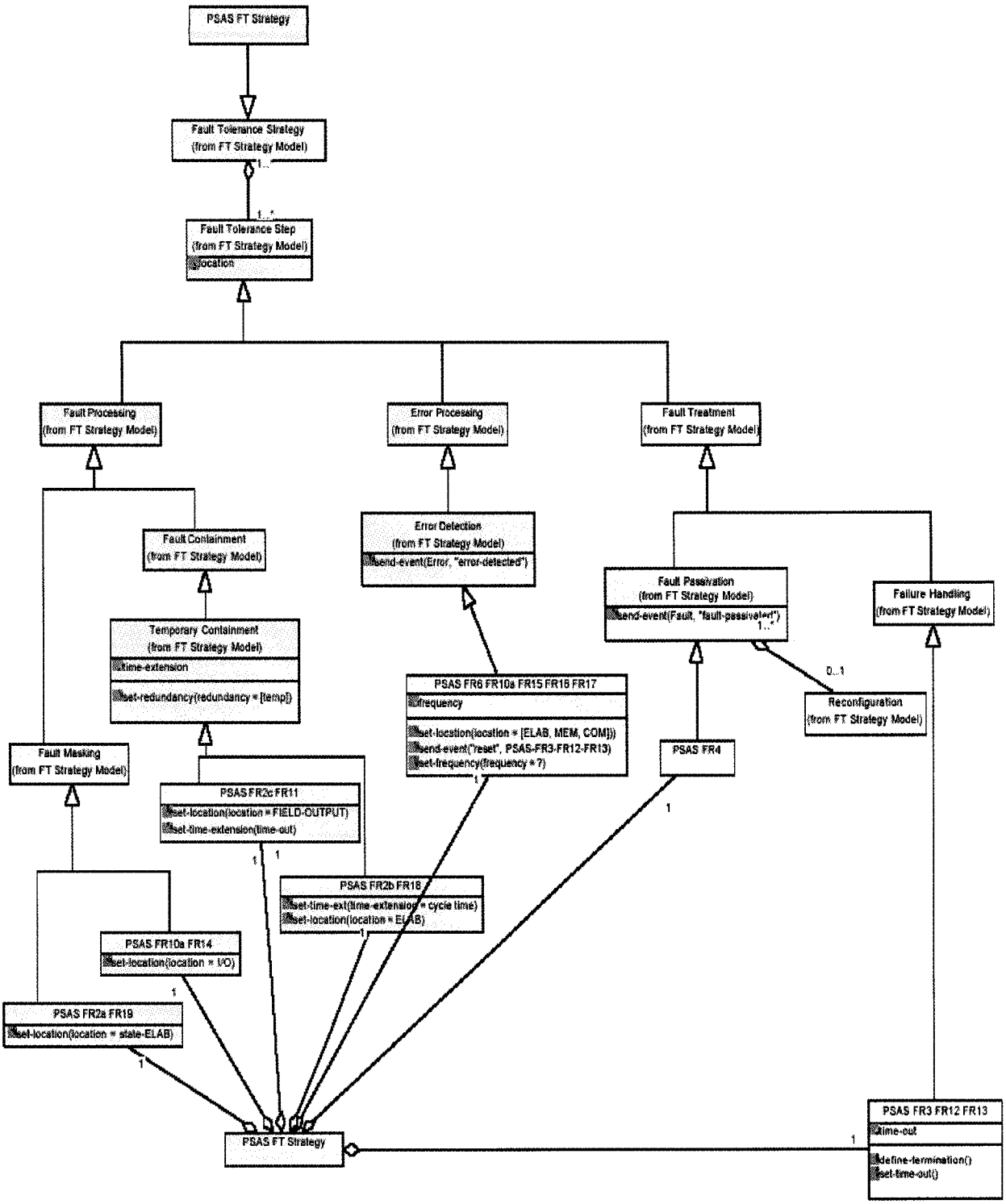

Figure 42. The main class diagram of the Package PSAS FT Strategy [TIRAN D1.1]

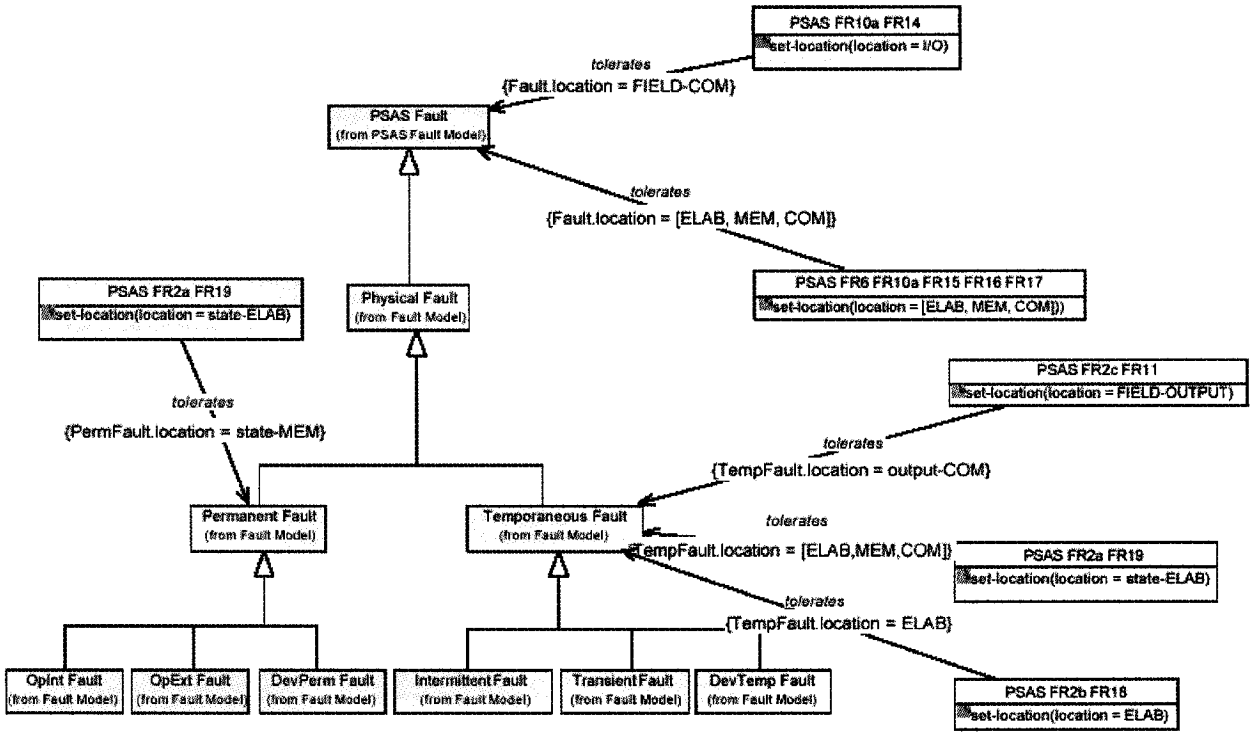

Figure 43. The class diagram Fault Model Relationships of the Package PSAS FT Strategy Model [TIRAN D1.1]

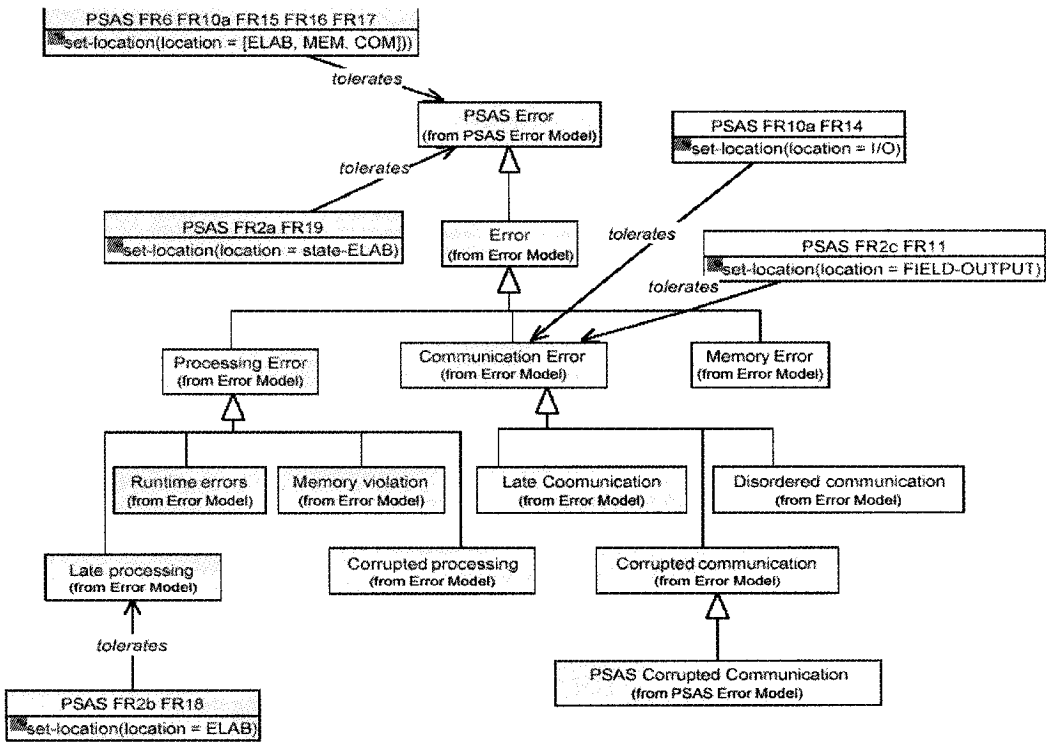

Figure 44. The class diagram Error Model Relationships of the Package PSAS FT Strategy Model [TIRAN D1.1]

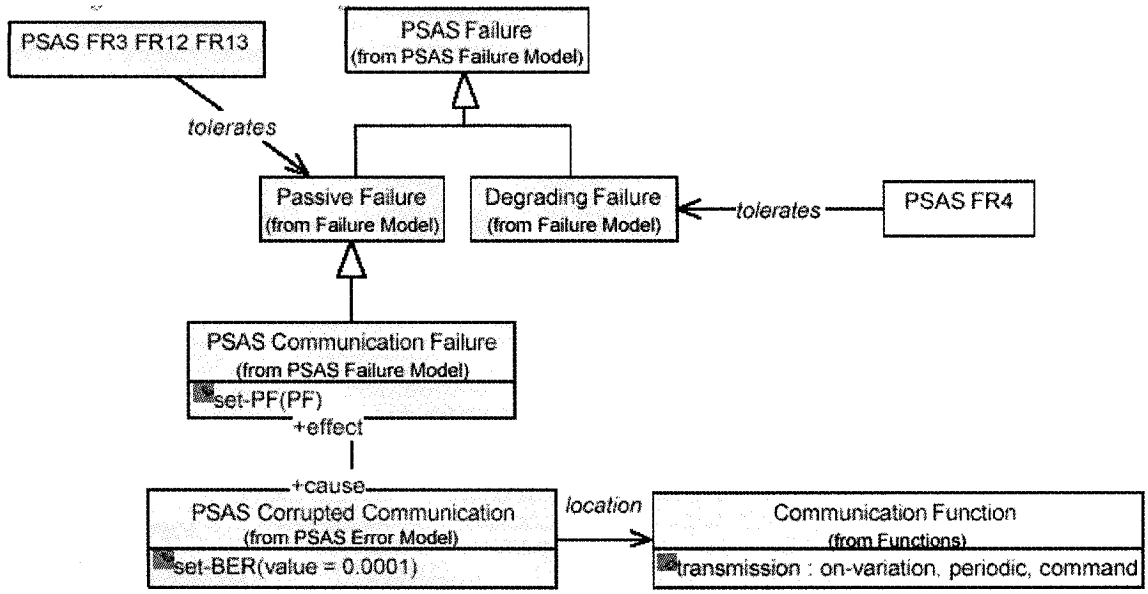

Figure 45. The class diagram Failure Model Relationships of the Package PSAS FT Strategy Model [TIRAN D1.1]

4.2 DepAuDE

Dependability for embedded **Au**tomation systems in **D**ynamic **E**nvironments with intra-site and inter-site distribution aspects (DepAuDE) is a project partially based on the ESPRIT project TIRAN discussed in the previous section. It is an IST (Information Society Technologies) project initiated in 2001 and completed in 2003. The partners were K.U. Leuven, CESI, Siemens, UniFe, UniTo, and TXT. It has been developed primarily for two target application areas: monitoring/control of energy transport and distribution, and distributed embedded systems. The pilot applications for DepAuDE include automated substations system and airfield lighting control system with sensor feedback.

The DepAuDE framework aims to provide "a methodology and an architecture to ensure dependability for non-safety critical, distributed, embedded automation systems with both IP (inter-site) and dedicated (intra-site) connections" [DepAuDE D8.6].

This part of the chapter presents a brief overview of the DepAuDE project. Section 4.2.1 discusses the requirements of the framework and Section 4.2.2 outlines the user support

provided by DepAuDE. Detailed information, public deliverables and publications on DepAuDE are available at [DepAuDE].

4.2.1 Framework Requirements

4.2.1.1 Shortcomings in TIRAN

The TIRAN approach concentrates on the computational aspects of achieving dependability. It considers intra-site connections only and assumes a reliable communication. The soft real-time support available in TIRAN is not adequate for all processes and it also lacks support for quality-of-service (QoS) levels.

The DepAuDE project aimed to develop a framework which satisfied the shortcomings in TIRAN. The pilot applications from the target application fields require that both intra-site and inter-site communications be supported.

4.2.1.2 Real-time Requirements

At the intra-site level, hard real-time requirements need to be considered. But, at the inter-site level, only soft real-time requirements can be handled because shared lines imply low predictability. To satisfy the intra-site real-time requirements, a real-time operating system (RTOS) is needed. For inter-site real-time requirements, DepAuDE should be able to support predictability and active backup of messages.

4.2.1.3 Quality-of-Service Requirements

With regards to quality-of service requirements, DepAuDE should consider the priority of the task and available resources, and accordingly offer quality of service. It should provide support for fault-tolerance techniques and recovery strategies. Remote maintenance and control should also be achievable. QoS for link failures may be improved by redundancy means.

4.2.1.4 Fault-tolerance Requirements

DepAuDE aims at tolerating physical faults and malicious faults. Malicious faults can occur in the context of inter-site communication. Physical faults can occur in computing

node components or network components. Inter-site connections include communication via Internet, communication via Intranet, and communication via a dedicated line. The DepAuDE framework should provide support for tolerating faults in communication and ensure dynamic reconfiguration and recovery in case of node failures and should adapt the recovery/reconfiguration strategies based on the resources available. To tolerate faults at the intra-site level, DepAuDE makes use of group communication. Finally, security support for inter-site communications should be available to avoid and tolerate malicious faults.

4.2.2 User Support

DepAuDE provides the user support available in TIRAN and in addition extends it to support inter-site communication in distributed systems.

4.2.2.1 Framework Support

DepAuDE is a fault-tolerance middleware framework that enables customization of fault-tolerance strategies and middleware according to the needs of the target applications. It supports the design of intra-site (client-to-backbone interface, ARIEL translator - discussed previously in Section 4.1.2.3) and inter-site mechanisms (gateway, UML specification). It implements a Basic Services Layer (BSL).

A distributed system developed according to the DepAuDE architecture is composed of a set of sites. Each site internally has nodes which represent processing units. They are locally connected by an intra-site communication network (local area network or dedicated lines). The sites are linked via an inter-site communication network (Internet, IP-based). A target application architecture is shown in Figure 46.

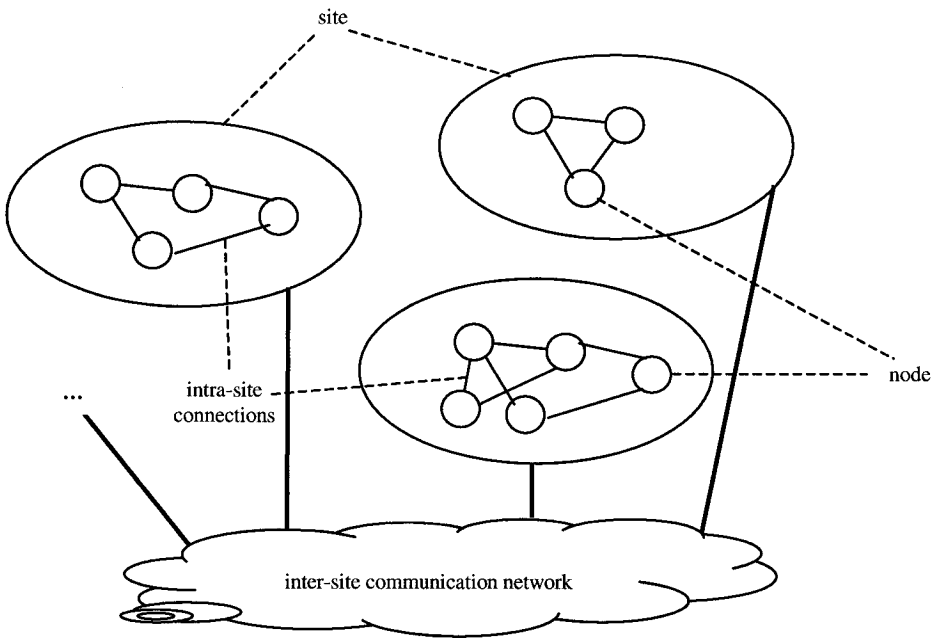

Figure 46. Target application architecture

4.2.2.2 Methodology Support

DepAuDE has a defined methodology which supports specification and validation of dependability requirements with the use of semi-formal and formal techniques using UML (Unified Modeling Language), TRIO (Tempo Reale Implicito), and GSPN (Generalized Stochastic Petri Nets). It also provides modeling support and predictive evaluation of the design. The DepAuDE methodology scheme is shown in Figure 47. The methodology support is similar to that outlined in TIRAN, with the inclusion of inter-site communication features for specification, validation, and modeling of requirements. Furthermore, the DepAuDE framework has been applied on the pilot applications to evaluate and show the feasibility of the framework. The results of this work offer useful guidelines for target users.

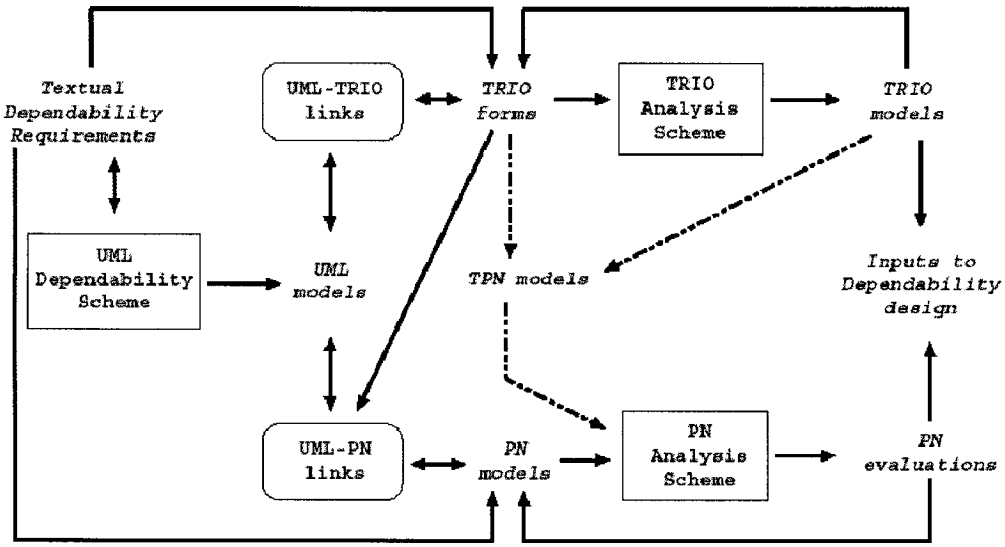

Figure 47. DepAuDE methodology scheme [GD2001]

4.3 TARDIS

The **Timely and Reliable Distributed Information Systems (TARDIS)** project was initiated jointly by Prof. Alan Burns of University of York (York) and A. M. Lister of University of Queensland (Australia) in 1990. The TARDIS framework was targeted towards avionics, process control, military, and safety critical applications. It was developed with the intention of creating a framework which considered non-functional requirements and implementation constraints from the early stages of software development.

The following sections describe the TARDIS framework starting from the requirements specification phase in Section 4.3.1 to logical and physical architectural design in Section 4.3.2. Section 4.3.3 briefly mentions the position of TARDIS with regards to software development methods. Finally, Section 4.3.4 presents a case study demonstrating application of the framework in the development of a pump control system. The pump control system has also been discussed in Section 3.2.5.

4.3.1 Requirements Specification

In addition to the functional requirements, the TARDIS framework takes into account three categories of non-functional requirements: dependability, timeliness, and dynamic change management. Dependability encompasses the requirements availability, reliability, safety, and security. Timeliness considers the requirements of responsiveness, orderliness, freshness, temporal predictability, and temporal controllability. Dynamic change management entails replacing parts or adding functionality to the system without halting it.

4.3.2 Architectural Design

The architectural design method followed in TARDIS has two phases: logical architecture and physical architecture. In this phase, consideration is given to issues of choices that can occur in the architectural design phase, like, a choice between replication and dynamic reconfiguration for improving reliability.

4.3.2.1 Logical Architecture

The logical architecture is involved with the design of the functional requirements. In this step, the object classes, interfaces, and relationships are defined without any consideration of the execution environment.

4.3.2.2 Physical Architecture

The physical architecture of the design is aimed at satisfying the non-functional requirements which are considered along with the functional requirements and constraints imposed on the execution environment. These constraints may include the network topology, the CPU clock speeds, resource availability, etc [FL92]. The software ultimately developed should be capable of being analyzed to see whether it satisfies all the non-functional requirements and the constraints.

In this step, instances of classes are considered and they are associated to the execution environment. The non-functional requirements are mostly incorporated as annotation or attributes to these classes and their methods.

For clarification, the architectural design method followed by TARDIS is demonstrated via a case study in Section 4.3.4.

4.3.3 TARDIS and Software Design Methods

TARDIS is a generic framework and does not impose any software design methods or languages on the developer. It maybe applied to any existing design methodology by following some specified rules (defined below) [FL92].

- It is required that the specification language be extended to consider non-functional requirements.
- A notation for specifying the characteristics of the execution environment must be available. The notation can be an extension of the specification language or a new addition.
- It is required that the design be incorporated with non-functional requirements and constraints introduced by the execution environment. The current design rules can be upgraded so as to produce a design that accounts for the aforementioned or a separate technique can be introduced to check validation of the design.
- It may be required to extend the design language, if non-functional obligations are still unsatisfied following architectural design.

[FL92] describes with an example how TARDIS can be applied together with a software development method, which uses the specification language *Z*, to develop a real-time data acquisition and display system. *Z* is extended to consider non-functional requirements, and the real-time logic (RTL) is used as the notation to define the target environment.

TARDIS is based on object-orientation, and hence the structural clash between the early stages and implementation is minimized. The variations that TARDIS introduces to the software development process are illustrated in Figure 48.

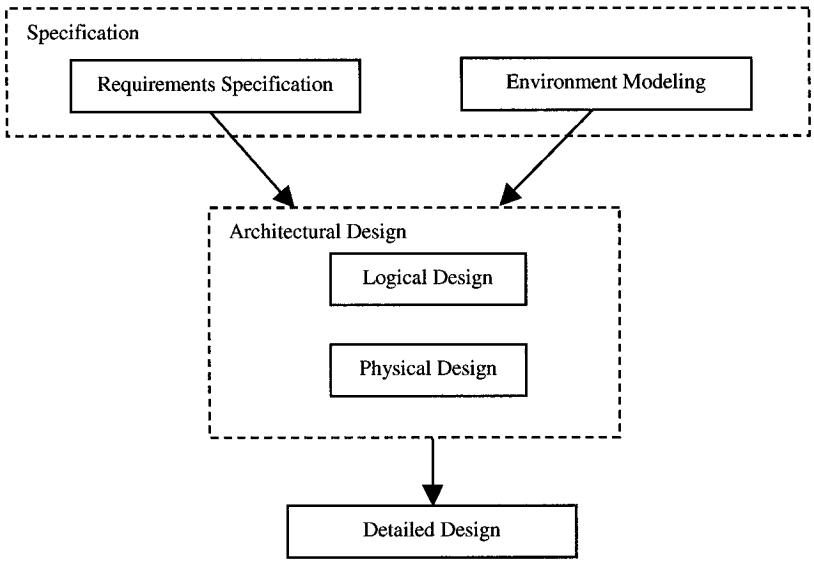

Figure 48. Software development using TARDIS

4.3.4 Case Study: Mine Control System

This section demonstrates the use of TARDIS with a case study. The application chosen is a standard in real-time systems literature: the pump (or mine) control system. The basic operation has been explained before in Chapter 3, Section 3.2.5, and is not repeated here.

4.3.4.1 Requirement Specification

In this section, the functional and non-functional requirements for the pump control system are presented as reported in [BL91a].

4.3.4.1.1 Functional Requirements

- *Pump operation.* The pump is switched on when the water level is below the high-water level and the methane level is below critical. In addition to automatic operation, the operator and the supervisor are allowed to switch the pump on and off based on some conditions. The operator is only allowed to switch on the pump when the water level is above the low-water level, and the methane level is below critical. The supervisor however can switch it on only based on the methane level, which has to be below critical.

The pump is switched off automatically when the water level goes below the low-water level or when the methane level reaches the critical level. The supervisor is allowed to switch it off only when the water level is below the high-water level.

- *Pump monitoring*. Every operation on the pump and its state alterations are logged.

- *Environment monitoring*. The environment sensors for methane and carbon monooxide gas, and airflow need to be constantly monitored and logged. The critical levels of these sensor values may lead to the pump being shutdown or to alarms being raised.

- *Operator information*. The operator should receive information about all critical readings of sensors.

4.3.4.1.2 Non-functional Requirements

4.3.4.1.2.1 Dependability

For the pump control system, the dependability requirements ensure that the system is reliable and safe.

Reliability of the pump system is measured by the number of shifts that can be allowed to be lost if the pump does not operate when it should be. In this case, a system can be said to be reliable if it loses at most 1 shift in 100. Also, even on pump failure, *a water accretion period* of one hour is allowed before a shift is defined as lost.

Safety of the pump system is related to the probability that an explosion can occur if the pump is operated when the methane level is above critical. In this case, the probability is assumed to be less than 10^{-7} during the lifetime of the system.

To achieve high levels of reliability, the pump system needs to be in operation at all required times. However, running the pump during unstable methane levels causes the safety level to fall. On the other hand, it would be safest if the pump was not operated at

all in abnormal circumstances, but this would make the system less reliable. Hence, there is a trade-off between reliability and safety here. But, it should be noted that safety is more desirable than reliability for such a system.

4.3.4.1.2.2 Timing

Monitoring periods are used to specify the maximum period environment sensors can be monitored or read. In this example, 60 second values have been used for all sensors.

A shutdown deadline is required according to which the pump is switched off when the methane level reaches the critical level. The relationship of the deadline (D) to the safety margin (M), the methane accumulation rate (R), and the sampling period (P) can be expressed by $R(P+D) < M$.

An operator information deadline should be specified which is the period by which the operator should be informed of critical gas levels.

4.3.4.1.2.3 Security

The security of the system is associated with the users accessing it. An operator should not get the rights of a supervisor when operating the pump.

4.3.4.2 Architectural Design

4.3.4.2.1 Logical Architecture

The logical architecture, as mentioned previously, considers the functional requirements of the system, and in this case also the security requirement. Hence, for this system, the functional requirements can be mapped to four classes: pump subsystem, data logger (introduced due to pump monitoring), environment subsystem, and operator. To satisfy the security requirement, the decision of having single or separate classes for the operator and supervisor should be considered. Preference is given to the latter since that way the authority of the user can be checked during compile-time.

The classes can be further decomposed, resulting in seven classes at the logical architecture phase: Pump controller, Pump, Water sensors, Environment monitor, Environment sensors, Operator, and Supervisor.

4.3.4.2.2 Physical Architecture

4.3.4.2.2.1 Dependability

At the subsystems level, safety of the system can be threatened due to the failures mentioned below.

"

> *the environment subsystem provides an incorrect (low) value for methane level when asked by the pump subsystem;*
>
> *the environment subsystem fails to generate an alarm signal when the methane level reaches the danger threshold;*
>
> *the communication medium fails to convey the alarm signal to the pump subsystem;*
>
> *the pump subsystem fails to switch off the pump when it receives the alarm.*

" [BL91a].

From the above, it can be deduced that safety of the system is dependent on the environment subsystem, the pump subsystem, and the communication medium between them. Two types of failures can affect safety: fail-silent and fail-noisy. The first step would be to create fault containment areas. The task of raising an alarm can be avoided, if the pump subsystem can be assigned an additional operation of checking the methane level continuously. This way the pump can switch itself off when it receives no response from the environment subsystem. This does not increase the design complexity. The system is now only affected by failures in a fail-noisy manner. In addition, time-stamping can be used when sending methane readings to enable the pump subsystem to realize when its getting old readings and act accordingly.

In the case of reliability, to prevent loss of shift, the pump should be repaired before the water accretion period passes.

Looking into more details, the component sensor in the pump subsystem needs to be annotated with attributes like failure probability and MTBF (mean time between failures). Since, sensors only fail in a fail-noisy way, replication of the sensors is required to tolerate hardware failure. Three sets of sensors can be used along with N-modular redundancy (NMR) technique for detecting and tolerating faults.

In a similar manner, the other components in the system can be analyzed and measures taken to achieve dependability. Due to space constraints, they are not described here but can be found in [BL91a].

4.3.4.2.2.2 Timeliness

- Periodicity - The operations which read the environment sensors (carbon monoxide and airflow) are assigned a period of 60 seconds. The pump however needs to get methane readings every 5 seconds. This period also to account for the methane sensor being read, the request sent by the pump controller and the delay caused due to communication. Realistic assumptions would probably estimate 1 second for communicating and 2 seconds each for the other two tasks.
- Deadlines – Operations that report critical events to the operator, and which switch off the pump due to unstable conditions, need to be assigned deadlines – 1 second each in this example.

It can be seen that the communication medium is a factor when determining timeliness of the system. For this reason, it is desirable to know the target environment beforehand so that appropriate considerations are made to it.

4.3.4.2.2.3 Distribution

As part of the physical architecture, the location of the objects in the execution environment needs to be realized also. The pump controller and the environment monitor need to interact frequently, and hence it is reasonable to locate them in the same processor to reduce communication delays. In order to tolerate hardware failures, these

should in turn be replicated on different processors. Similarly, decisions need to be taken regarding distribution of the other system components.

4.3.4.2.2.4 Communication

Another necessary design step is to decide on the communication mechanisms to be used, for example, remote procedure call (RPC) or asynchronous message passing. For the pump control system, RPC has been suggested, the reasons for which can be found in [BL91a].

4.4 Middleware Architectures

Middleware is software that is used to integrate heterogeneous software applications or products efficiently and reliably in a distributed computing environment. It is the middle layer between the application program and the platform and provides abstractions necessary for interfacing.

In [TD2001], Tirtea and Deconinck presented a survey of general middleware and their support for fault-tolerance. In [DepAuDE D2.1 & D2.2], the survey has been extended to include other middleware approaches that support fault-tolerance, but which are not in general use. This part of the chapter introduces these general (section 4.4.1) and fault-tolerant (section 4.4.2) middleware.

4.4.1 General Middleware

This section discusses the main general middleware available currently and their FT features: DCE, DCOM, Java RMI, and CORBA. DCOM, Java RMI, and CORBA are object-oriented, but DCE is procedure-oriented [DepAuDE D2.1 & D2.2].

DCE (Distributed Computing Environment) is an Open Group project developed to provide a software infrastructure for distributed computing. It has a set of programming interfaces and run-time services, and includes a comprehensive security model. DCE

supports fault-tolerance by replicating the core services provided by server programs [TD2001].

DCOM (Distributed COM) is a protocol developed by Microsoft Corporation as an extension of COM (Component Object Model). It enables software components developed and deployed by COM to communicate directly over a network. Fault tolerance support is limited, and is available at the protocol level. Network and client-side hardware failures can be detected via a pinging mechanism, and connections can be renewed automatically if the network recovers before the timeout period [TD2001].

Java RMI (Remote Method Invocation) is a communication mechanism which enables creation of distributed applications based on Java technology. Fault-tolerance support is available in Java RMI for reference counting in automatic memory management to protect against network failures. Also, Java has been extended to support real-time systems application programming [TD2001].

CORBA (Common Object Request Broker Architecture) is an OMG (Object Management Group) standard which offers an architecture and infrastructure that allows application programs to work together over networks irrespective of programming languages. TAO (The ACE ORB) implementation of CORBA supports fixed-priority real-time scheduling. Electra, another CORBA implementation, provides fault-tolerance with object replication. Real-time CORBA 1.0 supports QoS with standard policies and techniques [TD2001].

4.4.2 Middleware Architectures with FT Support

This section discusses some middleware approaches that support fault-tolerance.

Chameleon is an adaptive infrastructure, which supports multiple fault-tolerance strategies in a networked environment. Chameleon uses reliable agents that support user-specified levels of fault-tolerance. It considers satisfying dependability in terms of availability. Chameleon can be used for real-time applications with some additional

features added [DepAuDE D2.1 & D2.2]. A real-time application of Chameleon, a railway control system, is in operation.

ROAFTS is a middleware architecture providing real-time object-oriented adaptive fault-tolerance support. ROAFTS offers fault-tolerance schemes that can be applied to both process-structured and object-structured distributed real-time (RT) applications. These schemes are used to tolerate processor faults, communication link faults, interconnection network faults, and application software faults. ROAFTS is meant for implementation on COTS (Commercial Off-The-Shelf) and guarantees RT fault-tolerance when required [DepAuDE D2.1 & D2.2]

FRIENDS (Flexible and Reusable Implementation Environment for your Next Dependable System) is a software architecture which provides fault-tolerance and limited security support. It is built on subsystems and libraries of meta-objects. There is a fault-tolerance sub-system which incorporates fault-tolerance mechanisms for error detection, failure detectors, replication, reconfiguration, and stable storage. It does not provide specific support for real-time and quality-of-service requirements [DepAuDE D2.1 & D2.2]

AQuA (Adaptive Quality of Service for Availability) is an adaptive architecture for building dependable distributed systems. Fault tolerance is provided by Proteus, a dependability manager integrated into the architecture. Fault tolerance support is given to CORBA applications with replication of objects, and different levels of desired dependability and quality-of-service are provided. AQuA is capable of handling crash failures, value faults, and time faults. It incorporates means for detecting errors, treating faults, and reliable communication [DepAuDE D2.1 & D2.2]

TIRAN (Tailorable fault tolerance frameworks for embedded applications) is a software framework dedicated towards developing dependable systems in the automation systems domain. The TIRAN framework has been discussed in details at the beginning of this chapter and hence will not be repeated here.

4.5 Summary

The TIRAN framework [TIRAN D7.9], presented in Section 4.1, can be used as a cost-effective solution for developing dependable automation systems. It provides user support in the form of a methodology, which guides users to consider fault tolerance from the early software development stages. It ensures satisfaction of performance requirements from soft to hard real-time. The framework includes re-usable components as libraries of fault/error/failure mechanisms that can be customized for different applications. The framework is meant to act as a middleware for applications.

The TIRAN approach only supports embedded applications that are distributed locally (intra-site). The TIRAN project had left as future work two main things: extensions to provide support for globally (inter-site) distributed applications and to develop the TIRAN FT specification, validation, and verification methodology to support a widely distributed embedded applications.

Both of these issues have been considered in a follow-up project, DepAuDE (Dependability for embedded automation systems in dynamic environment with intra-site and inter-site distribution aspects) [DepAuDE] briefly discussed in Section 4.2 of this chapter. The DepAuDE project has improved the state-of-the-art in the addressed application fields. The methodology and architecture aims to develop more cost-effective and dependable systems with respect to other approaches available, particularly in terms of better QoS, maintainability and reusability.

Another noteworthy project, TARDIS [BL91a], separate from the above two has been discussed in Section 4.3 of this chapter. The TARDIS project can be considered to be inspired from HRT-HOOD. But where HRT-HOOD caters to real-time systems and considers satisfying only the *timing* non-functional requirement, TARDIS is targeted to a wider range of dependable systems and aims to satisfy other non-functional requirements like dependability also. But, the initial proposal of developing a framework presented in [BL91a] catered towards dependable systems was not completed. The project was,

however, continued in one area. Detailed work was done on the development of real-time systems using the TARDIS framework. [FL92] discusses the architectural design of non-functional requirements related to real-time issues using the specification language *Z* and RTL (real-time logic) in particular. Detailed design using TARDIS is considered in [BL91b] [FL92]. According to Fidge and Lister in [FL92], the TARDIS framework can also be applied to the design of systems where non-functional requirements like reliability, security, safety, fault tolerance, and system reconfiguration need to be satisfied.

The last section of this chapter, Section 4.4, presents a middleware survey as reported in [DepAuDE D2.1 & D2.2]. This survey was carried out as part of the TIRAN and DepAuDE project to research some available fault tolerance approaches and analyze the extent of fault-tolerance support provided in each. None of the approaches were seen to be adequate for developing applications in the target area and hence TIRAN and later DepAuDE were developed. Although, these two projects cater to a specific domain, to date they are two significant contributions to the development of dependable systems.

Chapter 5.

Other Fault-Tolerance Approaches

Although until today not much concrete work have addressed fault-tolerance, quite a few approaches have been proposed which address dependability requirements from the early stages of developments for various target domains. These vary from integration of exception handling features, to consideration of real-time issues in isolation, to providing support with an architecture, or framework which can be customized according to user needs. This chapter discusses a range of such approaches. Section 5.1 presents work on extensions of UML [UML2003] and Section 5.2 discusses some diverse ways of addressing fault-tolerance in the software process.

5.1 Extensions of UML

A few approaches have been proposed that use UML extensions to provide support for fault-tolerance. The extensions include mechanisms to incorporate stereotypes, tagged values, constraints, and profiles in the model. Two such approaches are discussed in this section.

5.1.1 Modeling Hard Real-time Systems with UML: The OOHARTS Approach

The Object-Oriented Hard Real Time System (OOHARTS) approach [KN99] aims to provide an object-oriented method to develop hard real-time systems. The process is based on UML [UML2003] and extension mechanisms related to timeliness, and hard real-time constructs of the HRT-HOOD [BW95]method (presented in Chapter 3).

The extensions made to UML are as follows:

- HRT class stereotype: Stereotypes are used to define the different kinds of real-time objects, and OOHARTS uses <<cyclic>>, <<aperiodic>>, <<protected>>, <<passive>>, and <<environment>>.
- Modeling of object behavior: A special form of UML statecharts, Object Behavior Chart (OBC), with means of representing timing constraints like deadline and period is used to define object behavior.
- Constraining the object synchronization: In addition to the UML communication stereotypes, OOHARTS adds <<aser>>, <<lser>>, <<hser>>, <<tm-hser>>, and <<tm-lser>>. They represent the same concepts as the *type of request* concept in HRT-HOOD.
- Constraining the objects concurrency: UML provides a concurrency attribute which can be of type sequential, guarded, or concurrent. OOHARTS extends this list by adding the stereotypes <<mutex>> (mutual exclusion), <<wer>> (write execution request), and <<rer>> (read execution request).

The OOHARTS method follows the traditional software development phases: requirements definition, hard real-time analysis, hard real-time design, and implementation. The requirements definition phase involves specification of both functional and non-functional requirements. OOHARTS differs from HRT-HOOD in two main ways: it is object-oriented, whereas HRT-HOOD is object-based, and OOHARTS introduces a software process with an additional phase, hard-real time analysis, which provides a framework for defining the structure and behavior of hard real-time systems using UML and the new extensions defined [KN99].

5.1.2 Developing Safety Critical Systems with UML

It is crucial when developing safety-critical systems to consider means to achieve the highest level of dependability. In [JJ2003], a method is presented which uses UML [UML2003] in the development of cost-effective and dependable safety-critical systems. The approach is based on using the UML extension mechanisms to incorporate safety requirements in a UML model. The mechanisms consider crash/performance failures and value failures which may cause message loss, delay, or corruption. Figure 49 shows the

stereotypes with the associated tags and constraints along with an informal description. For example, the stereotype <<risk>> can be used to describe a risk that arises in the physical level with the tag {failure} and <<error handling>> provides an object for handling errors in the subsystem level and is associated with the tag {error object}.

The approach also considers analyzing the UML model with a prototypical XMI-based tool to check if it satisfies the requirements. The tool is described in details in [JJ2003].

The approach discussed here considers non-functional requirements during the design phase in terms of safety. Jürgens has also proposed using UML to develop security-critical systems in [JJ2004]. Previously, he has also, in collaboration with others, described some approaches for systems development using UML which consider various criticality requirements.

Stereotype	Base Class	Tags	Constraints	Description
risk	link, node	failure		risks
crash/ performance	link, node			crash/performance failure semantics
value	link, node			value failure semantics
guarantee	link, node	goal		guarantees
redundancy	dependency, component	model		redundancy model
safe links	subsystem		dependency safety matched by links	enforces safe communication links
safe dependency	subsystem		«call», «send» respect data safety	structural data safety
critical	object	(*level*)		critical object
safe behavior	subsystem		behavior fulfills safety	safe behavior
containment	subsystem		provides containment	containment
error handling	subsystem	error object		handles errors

Figure 49. Stereotypes with associated tags and constraints [JJ2003]

5.2 Other Approaches

5.2.1 A Framework for Integrating Non-functional Requirements into Conceptual Models

In [CL2001], an interesting approach is presented addressing the need to capture non-functional requirements (NFR) at the early stages of development, by integrating NFR into conceptual models, specifically into the entity-relationship (ER) and object-oriented

(OO) models. The proposed method describes the use of the LEL (Language Extended Lexicon), discussed in [CL2001], and a NFR taxonomy to elicit the requirements. A comprehensive taxonomy of NFR is shown in Chapter 2. A LEL-NFR tool is required that captures terminologies relevant to the target field, referred to as the UoD (Universe of Discourse). This tool along with the NFR taxonomy is used to derive the NFR knowledge-base for a particular domain. These NFR are decomposed and represented in graphs which a slight variants of Chung's NFR graphs [CN2000]. Finally, the NFR are integrated into the conceptual models. In ER models, a NFR is shown in a rectangle with the UoD labeled over it, and connected to the relevant entity or relationship. In the OO model, NFR are added to class diagram by attaching two rectangles to the right bottom of the class with the UoD name in one and the NFR name in the other.

In [CL2004], this approach has been applied to UML, starting from use cases to class diagrams, sequence diagrams, and collaboration diagrams.

5.2.2 Exception Handling in the Development of Dependable Component-Based Systems

Exception handling is a technique that can be integrated into software systems to integrate the process of error recovery. In [RL2004], an approach is presented to handle exceptions in the development of dependable component-based systems. Rubira and others have proposed a way of incorporating the behavior of exceptions in the Catalysis process [DW98].

Catalysis is a methodology for developing object-oriented software based on the notion of components. The Catalysis method does provide a way of representing exceptions, but does not consider exceptional behavior during software development. Catalysis uses a concept called *collaboration,* which represents "a set of related actions between objects" [RL2004]. The proposed approach depends on this feature to identify the interactions occurring among components, which are structured as *idealized fault-tolerant components (IFTC)* (discussed in Chapter 2). It aims to extend Catalysis with provisions

for specifying exceptional behavior in the requirements phase, and then mapping them onto design and implementation.

To begin with, exceptions are added to use-case specifications in a formal manner as shown in Figure 50. The system is structured with IFTC and the propagation of exceptions is clearly modeled as illustrated in Figure 51. In the next phase, collaborations are defined from use-cases, where the pre- and post conditions are mapped to actions, which include refinements of the defined exceptions. A template is used to describe the collaboration, and class hierarchies of normal and exceptional behavior are produced. Following the design, ways to move on to implementation are suggested in [RL2004].

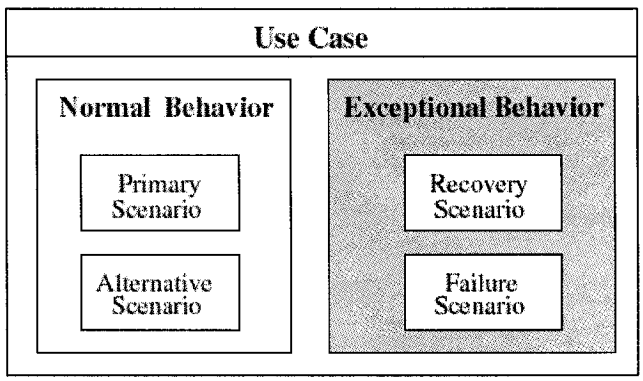

Figure 50. Structure of a use case [RL2004]

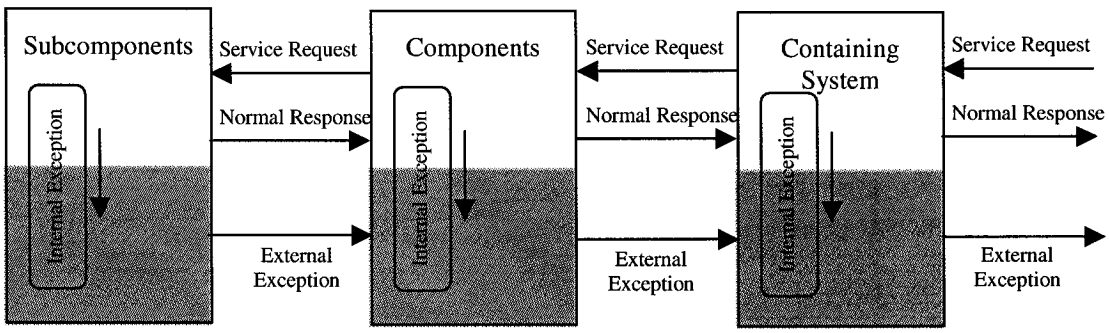

Figure 51. A software architecture composed by three IFTC [RL2004]

Although this approach makes use-case specification more complex, it provides a way for achieving dependability to some extent in component-based systems with the use of an error recovery mechanism.

5.2.3 EFTOS: FT Approach to Embedded Supercomputing

Embedded Fault-Tolerant Supercomputing (EFTOS) is an ESPRIT project completed in 1998, targeted towards industrial process-control, real-time applications, and embedded systems. It aims to provide a middleware framework to implement fault-tolerance to make embedded supercomputing applications more dependable.

The EFTOS approach overlaps with TIRAN (discussed in Chapter 4). Similar to TIRAN, it also follows a layered approach comprising of basic fault-tolerance tools and mechanisms, a backbone, and a high-level recovery language for specifying recovery strategies [DF2002]. The FT tools provided include a watchdog timer, a trap handler for exception handling, an atomic action tool, assertions, and a distributed voting mechanism. The framework which acts as a middleware between the application and the platform, can be customized according to the needs of the target application.

5.2.4 DELTA-4

Delta-4 [PB93] is another ESPRIT project which aims to provide an open architecture for development of dependable systems. It considers fault-tolerance and real-time issues, and is targeted at distributed real-time systems. The Delta-4 architecture is made up of software components located in host computers forming a possibly heterogeneous computer network. Each host computer together with a Network Attachment Controller (NAC) forms a node in the system. A NAC is a specialized communication processor which fails in a fail-silent manner.

Delta-4 tolerates hardware failures with the help of hardware and software redundancy. It supports active and passive replication of software components residing in homogeneous computers. Depending on the target application, replication can be used with or without voting. Voting mechanisms are included to address fail-uncontrolled hosts. In case of passive replication, systematic and periodic strategies for check-pointing are provided.

The original architecture was not suited for real-time application development. It has been extended with facilities to provide support for real-time systems. The

leader/follower replication strategy, discussed in [PB93], has been incorporated especially for use in real-time systems. Delta-4 addresses both soft and hard real-time behavior.

5.3 Related Work

Some related work considering fault-tolerance and other dependability attributes worth mentioning are listed below. Because of space constraints, it was not possible to describe them in details.

- MAFTIA (Malicious and Accidental Fault Tolerance for Internet Applications) is an European Union project completed in 2003 and is said to be the first project to address the need to tolerate malicious and accidental faults in large-scale distributed systems [MAFTIA].
- GUARDS (Generic Upgradeable Architectures for Real-Time Dependable Systems) [PA99] is an ESPRIT project aiming to provide methods, techniques, and tools for design, implementation, and validation support in safety-critical real-time systems.
- MARS (Maintainable Real-Time System) [RL95] is an architecture specialized for time-triggered applications, and addresses fault-tolerance with active replication means and other hardware FT measures to satisfy hard real-time requirements.
- Aurora Management Workbench provides a software framework for developing reliable, scalable, and configurable distributed applications [AMW].
- DOORS is a framework developed to provide support for building fault-tolerant applications in CORBA [GN2000].
- HIDE (High-level Integrated Design Environment for Dependability), an ESPRIT project, addressed the need for early validation of UML-based design [BC2001]. In [MC2003], Chin proposes an approach, as part of HIDE, to extend UML towards a useful OO-Language for modeling dependability features. It provides abstractions to incorporate common dependability requirements in the UML model.
- In [GR2003], a fault-tolerant software architecture for component-based systems is proposed. It is based on IFTC (idealized fault-tolerant components), which enables it to handle software faults, providing higher levels of dependability.

Chapter 6.

Survey Results

Several approaches with varied dependability and fault-tolerance support have been discussed in this thesis. The approaches have been evaluated based on several criteria: non-functional requirements (NFR) that are addressed, fault-tolerance features that are offered, application environments in which the approaches can be used, and NFR specification support. The comparisons are presented in Section 6.1, 6.2, 6.3, and 6.4 respectively. In addition, Section 6.5 shows a middleware comparison taken from [DepAuDE D2.1 & D2.2], and Section 6.6 compares FT design approaches based on the report in [TIRAN D3.3].

Note: For approaches where adequate information was not available to state dependability or fault-tolerance support, the symbol " – " has been used. Also, the approach, a framework for integrating non-functional requirements into conceptual models, *presented in Chapter 5, Section 5.2.1, is too general and hence has not been considered in the comparisons in this chapter.*

6.1 Non-functional Requirements

This section shows a comparison of the approaches discussed in this thesis based on some important non-functional requirements. The requirements considered have been introduced in Chapter 2, and include *dependability*, *timeliness*, *adaptability*, and *quality-of-service (QoS)*. Dependability, as defined in Chapter 2, refers to availability, reliability, safety, confidentiality, integrity, and maintainability. Availability and reliability are related attributes and can be classified as "avoidance or minimization of service outages" [AL2000]. Also, a specialization of availability and integrity with respect to authorization, together with confidentiality can be grouped together as the *security* requirement [AL2001]. The approaches have been evaluated based on the requirements

that are satisfied or taken into consideration (marked with ✓), and a comparison is illustrated in Table 6. The lack of support is shown with the × symbol.

	Availability/ Reliability	Safety	Security	Maintainability	Timeliness	Adaptability	Quality-of-Service	Comments
HOOD	×	×	×	✓	✓	×	×	limited maintainability (only exception handling); timeliness (only SRT);
HRT-HOOD	✓	✓	×	✓	✓	✓	×	limited maintainability (only exception handling and maybe replication); adaptability (mode changes);
TIRAN	✓	✓	×	✓	✓	✓	×	assumes reliable communication; safety (only by criticality level)
DepAuDE	✓	✓	✓	✓	✓	✓	✓	considers intra- and inter-site communication; safety (only by criticality level)
TARDIS	✓	✓	✓	✓	✓	✓	×	not targeted to specific NFR – open framework
OOHARTS	✓	✓	×	✓	✓	✓	×	limited maintainability (exceptions); adaptability (mode changes);
EFTOS	✓	×	✓	✓	✓	✓	×	security (integrity); timeliness (esp. SRT);
DELTA-4	✓	×	✓	✓	✓	×	×	user-specified level of dependability; maintainability (only replication);
Chameleon	✓	×	×	✓	×	✓	×	supports different levels of availability requirements; adaptability (mode changes);
ROAFTS	✓	✓	×	✓	✓	✓	×	guarantees RT FT; adaptability (mode changes); survivability;
FRIENDS	✓	✓	✓	✓	×	×	×	security (communication);
AQuA	✓	×	×	✓	×	✓	✓	user-specified level of availability;
JJ[4]	×	✓	×	×	×	×	×	targets specific dependability requirements
RLFF[5]	×	×	×	✓	×	×	×	targets specific dependability requirements

Table 6. Comparison based on NFR

6.2 Fault Tolerance Features

Table 7 presents a comparison of the fault-tolerance support in each approach. The classification is based on the failure domain support, error processing support, and fault treatment support. In addition, the important techniques for error processing and fault treatment considered in each approach have also been mentioned.

[4] **J**an **J**ürgens: *Developing safety-critical systems with UML.*
[5] C. M. F. **R**ubira, R. de **L**emos, G.R.M. **F**erreira, F. Castor **F**ilho: *Exception handling in the development of dependable component-based systems.*

	Failure Domain		Error Processing	Fault Treatment	Means
	Value	Timing			
HOOD	×	×	detection	no support	exception processing, deadlock avoidance techniques
HRT-HOOD	×	✓	detection	reconfiguration	exception processing, replication
TIRAN	✓	✓	detection, localization, recovery	diagnosis, masking confinement, dynamic reconfiguration, graceful degradation	exception handling, design diversity, stable memory, watchdog, local voter, distributed synchronization, time-out, standby sparing, recovery blocks, NMR
	computing failures only				
DepAuDE	✓	✓	same as above	same as above	same as above & group communication
TARDIS	✓	✓	detection, recovery	diagnosis, isolation, confinement, reconfiguration	timeout, HW/SW repair/replacement, NMR, failure messages
OOHARTS	-	✓	detection	-	exceptions, timeout, deadlock avoidance techniques
EFTOS	✓	✓	detection, isolation, recovery	masking, fault-tolerance	exception handling, watchdog, atomic actions, distributed voting, recovery language
DELTA-4	✓	✓	detection, recovery	reconfiguration, fault-tolerance	active/passive/leader-follower replication, voting, timeout
Chameleon	✓	✓	detection, recovery	masking, system reconfiguration, failure recovery	reliable agents, TMR (H/W), checkpoints, voting (distributed & majority)
ROAFTS	✓	✓	detection, recovery	fault-tolerance	watchdog timer, predictable communication, process scheduling, backward recovery (SRT), forward recovery (HRT), recovery blocks; active replication
FRIENDS	physical crash failures only		detection, recovery	reconfiguration, fault-tolerance	leader-follower-replication, stable storage, primary backup, failure suspectors, group communication
AQuA	✓	✓	detection, recovery	fault-tolerance, system reconfiguration	active/passive replication & degree, voting, monitors, group communication
JJ	✓	✓	detection, containment, handling	compensation	space/time/information redundancy, voting
RLFF	-	-	detection, recovery	no support	exception handling, IFTC

Table 7. FT support

6.3 Target Domain

This section categorizes the approach by their type: method, framework, middleware, architecture, or technique that targets specific dependability requirements. Table 8 shows this classification along with the target domain and environment the approach was developed for.

		Target Domain	Application Environments
Method	HOOD	embedded real-time systems	space, energy, defense, transport
	HRT-HOOD		avionics
	OOHARTS	hard real-time systems	-
Framework/ Middleware	TIRAN	embedded automation systems	automation (energy, transportation)
	DepAuDE	non safety-critical systems, distributed embedded systems	automation
	TARDIS	safety-critical embedded systems	avionics, process control, military
	EFTOS	soft real-time, mission-critical, embedded supercomputing systems	process control, signal processing
	ROAFTS	safety-critical, distributed real-time systems	broad range
Architecture	DELTA-4	distributed, real-time systems	computer-integrated manufacturing, process control, office automation
	Chameleon	real-time, networked systems	broad range
	FRIENDS	distributed systems	general
	AQuA	distributed systems	-
Technique	JJ	safety-critical systems	fly-by-wire in avionics, drive-by-wire in automotive, etc.
	RLFF	component-based systems	-

Table 8. Approaches and their target environment

6.4 Support for NFR Specification

The NFR specification support or lack of it in each of the previously mentioned approaches is briefly stated below.

HOOD does not provide guidelines for NFR specification. However, it is possible to define such requirements at an early stage.

HRT-HOOD and OOHARTS provide limited support for NFR specification – only timing requirements, like periodicity and deadlines, can be defined.

TIRAN and DepAuDE provide a methodological support for the collection, specification, and validation of NFR from the initial stages of development.

TARDIS enables specification of dependability, timeliness, and dynamic change management requirements but concrete guidelines are only available to specify real-time issues.

EFTOS supports integration of NFR by providing a library of FT tools, which can be adapted according to the application needs.

DELTA-4, Chameleon, and AQuA allow the user to specify NFR in terms of the level of dependability required. For example, in AQuA, it is possible to specify the QoS level. They provide limited FT techniques, which the user can choose from.

ROAFTS is a middleware architecture, which offers fault-tolerant schemes that can be adapted but does not provide a methodological support to specify NFR in an application. Similarly, in the FRIENDS architecture, no guidelines are provided for NFR specification, and it is the user's responsibility to use the scheme appropriately.

The approach, developing safety-critical systems with UML, considers NFR from an early stage and provides support to incorporate such requirements in the UML models. The other approach by RLFF only enables specification of exceptions and exception handling in the use cases.

6.5 Middleware Comparison

This section presents a comparison of the middleware, discussed in Section 4.4, with respect to their fault-tolerance (FT), real-time (RT), and quality-of-service (QoS) support. This comparison, presented in Table 9, is based on a study carried out in the DepAuDE project [DepAuDE D2.1 & D2.2]. The symbol * is used to denote the level of fault-

tolerance, real-time, and quality-of-service support provided by each middleware, with ***** being the highest level.

FT	**CORBA** ***	different implementations provide fault tolerance: e.g. Electra supports fault tolerance using object replication
	Java RMI *	fault tolerance in memory management for automatic garbage collection
	DCE ***	core services replicated
	DCOM *	support fault tolerance at protocol level for detecting network and client-side hardware failure
	Chameleon *****	user-specified level of fault tolerance; a special component designed for fault tolerance (Fault Tolerance Manager)
	ROAFTS *****	design to support adaptive fault tolerance; tolerates processor faults, communication link faults, application software faults
	FRIENDS ****	fault-tolerant dedicated subsystems (FTS, GDS)
	AQuA *****	fault-tolerant dedicated component: Proteus; Proteus tolerates crash, value and time faults
	TIRAN *****	adaptive, based on linguistic approach; ARIEL-language for error recovery
RT	**CORBA** ***	Real-Time CORBA 1.0 with fixed priority scheduling
	Chameleon ***	railway control system - an RT Chameleon application with a special component design for real-time support (Real Time Manager)
	ROAFTS ****	design to guarantee real-time fault tolerance; uses the Time-triggered Message-triggered Object (TMO) model
QoS	**CORBA** ***	Real-Time CORBA 1.0 facilitates end-to-end predictability
	AQuA ****	Quality Objects (QuO), components of AQuA allow to specify quality-of-service requirements at application level
	TIRAN **	supply different strategies for degradation of services

Table 9. Middleware comparison [DepAuDE D2.1 & D2.2]

6.6 Comparison of Designs of FT Approaches

This section presents a comparison of some approaches to provide fault-tolerance, based on the description in [TIRAN D3.3]. The approaches include the system approach (used in DELTA-4 and GUARDS), library approach (used in Project Isis), metaobject protocols and reflection (used in FRIENDS), and the recovery language approach (used in EFTOS), and an approach based on the library and recovery language approach (used in TIRAN and in DepAuDE). Table 10 shows a summary of the comparison based on the description in [TIRAN D3.3].

Attributes / Approaches	Efficiency	Transparency	Portability	Cost of Adoption	Flexibility	Examples
System Approach	High	High	Medium	Low	Low	Good for HRT Systems; No FT support within target application; Used in Project DELTA-4 and GUARDS
Library Approach	Medium	Medium	High	Low-to-medium	Medium	Not good for HRT systems; Used in Project Isis
Metaobject protocols and reflection	Medium	High	Medium-to-high	Medium	Medium	Good for object redundancy but not for distributed recovery blocks and such; Used in FRIENDS
Recovery Language Approach	Medium	Medium-to-high	High	Low-to-medium	Medium	Developed in the EFTOS framework
Combination of library and recovery language approach	Medium	Medium-to-high	High	Low-to-medium	High	TIRAN, DepAuDE

Table 10. Comparison of different orthogonal approaches

Chapter 7.

Future Work

In today's complex systems, esp. real-time systems, software availability and reliability are crucial. However, although fault tolerance techniques have been available for over 30 years now, their presence in the software development process is insignificant. Current mainstream software engineering methods do not consider fault tolerance in the requirements engineering stage, and only sometimes much later in the development cycle. But, most modern systems must provide or can at least benefit from some form of fault-tolerance. In [JL96], Laprie proposes a model that incorporates the dependability processes, fault forecasting, fault removal, fault tolerance, and fault prevention, into the software development lifecycle. The model is illustrated in Figure 52.

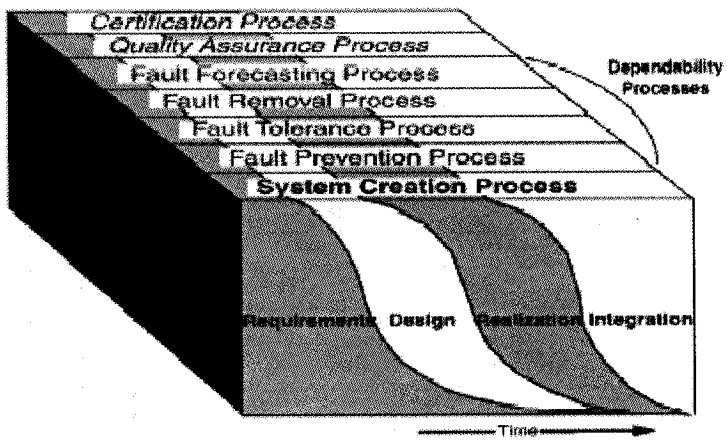

Figure 52. Dependability-explicit development model [KL2000]

It is obvious from the literature overview in this thesis that only limited research has been conducted in this direction. Quite a few projects did start with similar goals, but died before much progress was made. Hence, there are still many open questions and possibilities for major contributions. One significant work would be to to integrate the concern of fault tolerance into the software development cycle. The resulting benefits of this work are obvious. Having fault tolerance in mind from the beginning allows software

developers to "engineer" support for fault tolerance. It makes it possible to state the required level of fault tolerance precisely and then choose the appropriate models, design and infrastructure to achieve it. It helps in reducing the complexity of fault tolerance, results in clearer program code, increased readability, less maintenance overhead, and delivers adequate performance. The concrete output would be to add fault tolerance support to a mainstream development method, e.g. the Rational Unified Process. The work can be based on the Unified Modeling Language and the Object Constraint Language.

It involves two essential tasks: the requirements engineering and analysis, and the architecture and design. The requirements engineering and analysis phases of software development methods concentrate on elaborating a concise and complete specification of the application under development. The specification is considered to be the definition of correct program behavior. At this stage, integration of fault tolerance means identifying the need for fault tolerance, finding the places where it is needed, and specifying what degree of fault tolerance must be achieved.

In the architecture and design phases, a solution that provides all the services specified in the analysis phase must be devised. Object-oriented software development methods assist the designer in this task by assigning functionality to objects based on responsibility, and design patterns help to determine elegant interactions among objects. At this stage, the design process must be augmented in such a way that it leads to a solution that additionally fulfills the fault tolerance requirements. For this purpose, error confinement techniques provided by well-known fault tolerance models, forward and backward error recovery, and structured exception handling can be used. The resulting design should also be able to make use of fault tolerance infrastructures.

The results of this work would indeed be of great use to software developers for producing cost-effective dependable systems.

Chapter 8.

Conclusion

This thesis work started with the goal of identifying software processes that address fault tolerance in the early development stages. Fault tolerance has been stated as a means to satisfy non-functional requirements, mainly availability, reliability, safety, security, timeliness, adaptability, and quality-of-service. Hence, it was surveyed which approaches aim to fulfill such requirements.

In this thesis, numerous approaches are presented. The approaches are an assortment of methods, frameworks, middleware, architectures, and techniques. The software development methods discussed, HOOD and HRT-HOOD, consider real-time issues in isolation. Unfortunately, no methods are publicly available that integrate dependability concerns into the software development life cycle.

However, some middleware frameworks directed towards specific domains are available, namely TIRAN, DepAuDE, and TARDIS. These approaches consider dependability requirements like reliability, availability, security, timeliness among others, and provide a methodological support for customizing the framework to the application needs. TIRAN and DepAuDE make use of UML elements like *packages* and *class diagrams* to build models and specialized languages to enable user-specified recovery strategies. Other middleware like EFTOS and ROAFTS provide tools, which the user can adapt accordingly. Software architectures, like DELTA-4, Chameleon, FRIENDS, and AQuA, have attempted to satisfy hardware fault tolerance requirements to some extent by supporting techniques like replication, but they do not provide guidelines to the user or facilitate making design decisions during software development.

Other approaches have originated based on methods like HRT-HOOD, and processes like Catalysis. Several have proposed extension mechanisms for UML to integrate non-functional requirements into the design models.

This survey shows that there is a lot more work to be done in order to make fault tolerance an integral part of software development. In particular, there is close to no methodological support or guidance available for developers for achieving desired levels of fault tolerance. The approaches mentioned above are all domain-specific. Some approaches target distributed systems, but most of them are intended especially for embedded real-time systems. However, a mainstream software development method that integrates fault tolerance into the software development life cycle remains to be established. Fault tolerance concerns should be addressed earlier in the software development life cycle. Clear guidelines helping developers to choose the right software architecture and models supporting fault tolerance should be established. Finally, mappings of the models to fault-tolerant middleware or programming constructs have to be defined. Further research and concrete work in this field is essential to achieve cost-effective dependable systems.

Appendix A

A list of non-functional requirements [CN2000]

accessibility
adaptability
affordability
availability
capacity
cohesiveness
communication time
comprehensibility
confidentiality
controllability
correctness
customer evaluation time
data-space performance
dependability
distributivity
domain analysis time
enhanceability
extensibility
feasibility
generality
impact analyzability
inspection cost
inter-operability
learnability
maintenance cost
mean performance
nomadicity
operability
perfomability
planning time
precision
productivity
promptness
reconfigurability
reengineering cost
replaceability
risk analysis cost
safety
security
simplicity
space boundedness
stability
supportability
susceptibility
testing time
timeliness
trainability
usability
variability
visibility

accountability
additivity
agility
buffer space performance
clarity
commonality
compatibility
conceptuality
configurability
coordination cost
cost
customer loyalty
decomposability
development cost
diversity
efficiency
evolvability
external consistency
flexibility
guidance
independence
inspection time
internal consistency
main-memory performance
maintenance time
measurability
observability
operating cost
performance
plasticity
predictability
project stability
prototyping cost
recoverability
reliability
replicability
risk analysis time
scalability
sensitivity
software cost
space performance
standardizability
surety
sustainability
throughput
tolerance
uniform performance
user-friendliness
verifiability
wrappability

accuracy
adjustability
auditability
capability
code-space performance
communication cost
completeness
conciseness
consistency
coordination time
coupling
customizability
degradation of service
development time
domain analysis cost
elasticity
execution cost
fault-tolerance
formality
hardware cost
informativeness
integrity
intuitiveness
maintainability
maturity
mobility
off-peak period performance
peak-period performance
planning cost
portability
process management time
project tracking cost
prototyping time
recovery
repeatability
response time
robustness
secondary-storage performance
similarity
software production time
specificity
subjectivity
survivability
testability
time performance
traceability
uniformity
validity
versatility

Appendix B

The FT requirements listed below have been taken from [TIRAN D1.1].

A category identifier followed by a progressive number labels each requirement. Four category identifiers have been considered: *SR* (System Requirement), *DR* (Dependability Requirement), *FR* (Fault Tolerance Requirement), and *TR* ((real) Time Requirement).

SR1: "At the most highest level the automated system is composed by two sub-systems, namely the Primary Substations Automation System (i.e. the PSAS automation system) and the Primary Substation of the Energy Distribution Network (i.e. the PS plant). Primary Substations (PS) consist of Busbars, Switches, Insulators, Transformers and Capacitors. The PSAS architecture includes a Local Control Level (LCL) and a number of Periphery Units (PU) distributed on the plant, each PU being associated with a plant component. LCL in turn consists of a main Elaboration Unit plus a number of Communication Units."

SR2: "Each PU provides the following functions at the component level: primary and secondary protection levels, monitoring, command & control, diagnostic, data measurement, interface with the local operator, communication with the LCL. The LCL provides the following centralised functions at the PS level: monitoring, command & control, PU supervision, additional protection ('spare' level) interface to remote control systems and to local and remote operators."

TR1: "More than a single cycle-time for the whole application, specific real-time requirements can be given for the main PSAS functions, summarised as follows 20:

elaboration function	Texec
explicit cmd	<1 s
automatic cmd	<0.2 s
primary protection	<0.1 s
secondary protection	<0.3 s
'spare' protection	<1 s
'spare' protection (reaction to special events)	<20 ms
interface with centre	<1 s
diagnostic	<60 s
new interaction functions with UP	< 60 s

communication bandwidth	
LCL-PU (state-measures) serial	20 kbit/s
LCL-centre Ethernet	10 Mbit/s

."

DR1: "The ongoing development of new generation PS defines different availability values associated to system components depending on their criticality level. Reference values for the (minimal) availability follow:

- LCL-Elaboration Unit: MTBF = 100000-125000 hours
- LCL-Communication Units: given the high criticality of their associated functions, they require an higher availability than the Elaboration Unit; MTBF = 200 years
- PU: given the high number of PU contained in a PS, MTBF = 66 years."

FR1: "Faults to be taken into account for the PSAS concern permanent, intermittent and transient faults affecting system physical components. Most of transient faults are caused by electromagnetic interference."

FR2: "Corruption on input/output, elaboration, memory and internal communications, caused by any first fault (transient, intermittent or permanent) must be tolerated

a) allowing to preserve a working state acceptable and coherent with the history of the system

b) avoiding or handling any loss of control

c) avoiding to transmit wrong output to the plant, the operator, the remote systems."

FR3: "If the erroneous situation can not be recovered according to required mode and within given time constraints, then the disconnection of the automation system from the plant (auto -exclusion) must be guaranteed, leaving the plant in an acceptable state, forcing the output to assume a pre-defined secure configuration, providing appropriate signaling to the operator and to the remote systems (as automation system failures should not affect the plant)."

FR4: "Graceful degradation must be possible according to what allowed by system functions, involving the exclusion of faulty components (waiting for a maintenance intervention) and allowing the correct operation of remaining components."

FR5: "The accepted bound for the Probability of non recovered error (PE) within a transmission of data from a source to a destination, assuming a Bit Error Rate lower or equal to 10- 4, are given in the table

Data Type	**PE**
Periodically transmitted	<= 10- 6
Transmitted on variation	<= 10- 10
Commands	<= 10- 14

."

FR6: "Timing and mode of detection and recovery of errors due to the first fault must allow to minimize the probability of the occurrence of a second fault before a complete handling of the first one (fault overlap)."

FR7: "Results of error processing and fault treatment must be made available to the operator during system operation and recorded for later access - e.g. by signals and chronological event recording."

FR8: "System configuration data and running code must be non corruptible."

Containment and masking of undesired effects

FR9: "Error propagation through system components must be avoided by applying appropriate confinement techniques."

FR10: "Communication redundancies:

a) I/O circuits from/to the field should contain redundancies which allow masking faults on any duplicated I/O point; these redundancies shall be diagnosed.

b) Internal communications, i.e. between the LCL and each PU as also among the LCL components, should rely upon a redundant (double) dedicated LAN, based on FDDI technology."

FR11: "Undesired effects on output, due to a fault on any system component must be confined by applying appropriate delays (e.g. by means of hw filters) whose dimensioning shall allow the prompt intervention of the auto -exclusion circuit."

FR12: "A mechanism for the auto-exclusion of the system (Watch-Dog) should be provided which, if not reset before the expiration of a pre-fixed time -out, disconnects the system from the plant, forcing the output to assume a predefined secure configuration."

FR13: "The auto-exclusion should guarantee a high availability, integrity and security - e.g. by a redundant and periodically tested auto-exclusion mechanism, with auto-diagnostics (e.g. a test tolerant watch-dog)."

FR14: "The adoption of masking techniques on signals to/from the plant, the operator and the remote systems could be needed for some essential or particularly disturbed communications (e.g. auto-correcting code based on information redundancy, re-transmissions), if not already supported by adopted communication protocols."

Detection of erroneous situations

FR15: "The system should perform a periodic auto-diagnostic activity on its main components at an appropriate frequency, organised into diagnostic chains of on-line tests (e.g. complete CPU test, followed by addresses and data test, followed by RAM/ROM test and finally test of I/O circuits):"

FR16: "In absence of diagnosed errors, the auto -diagnostic activity will be followed by a reset command to the auto -exclusion mechanism of the system."

FR17: "A frequent and wide diagnostic (extended to each critical component) is fundamental at the aim of decreasing the probability of overlapping faults as much as possible (e.g. at each elaboration cycle)."

Recovery of erroneous situations

FR18: " Temporary confinement of blocking situations should be guaranteed:

a) system evolution must be guaranteed.

b) any loss of control (e.g. livelock, deadlock) must be confined within a single application cycle - e.g. by cyclic restart, which allows to restart erroneous components (e.g. tasks).

c) it is necessary to avoid that errors caused by non detectable temporaneous faults could became permanent, e.g. flips of registers or memory cells caused by EMI which has overcome protecting barriers - e.g. by cyclic restart, which allows to restart hw components from their own initial state."

FR19: "Proper evolution according to the system history needs to be guaranteed:

a) The evolution must be guaranteed between acceptable states.

b) Leaving an acceptable state must be allowed only towards another acceptable state - e.g. by using mechanisms which maintain the current state (judged correct) till a confirmation of correctness of the next state (e.g. by using a Stable Memory).

References

[AK87] P.E. Ammann, J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance". In *Proceedings of the 17th International Symposium on Fault–Tolerant Computing Systems (FTCS–17)*, Pittsburgh, PA, 1987, pp. 122–126.

[AK88] P.E. Ammann, J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance". In *IEEE Transactions on Computers* **37**(4), 1988, pp. 418–425.

[AL2000] A. Avizienis, J.-C. Laprie, et al., "Dependability of computer systems: Fundamental concepts, terminology, and examples". In *Proc. 3rd IEEE Information Survivability Workshop (ISW-2000)*, Boston, Massachusetts, USA, October 24-26, 2000, pp. 7-12.

[AL2001] A. Avizienis, J.-C. Laprie and B. Randell, "Fundamental Concepts of Dependability", Technical Report, CS-TR: 739, Department of Computing Science, University of Newcastle, 2001.

[AL81] T. Anderson, P.A. Lee, "Fault Tolerance - Principles and Practice", Prentice Hall, Englewood Cliffs, NJ, 1981.

[AL92] A. Lister, "Design of dependable real-time systems". In *Proc. of the 14th Intl. Conf. on Software Engineering*, 1992, pp. 35-36.

[AMW] R. Buskens, A. Siddiqui, et al., "Aurora Management Workbench", Bell laboratories, 2003, http://www.bell-labs.com/project/aurora.

[BC2001] A. Bondavalli, M.D. Cin, et al., "Dependability Analysis in the Early Phases of UML Based System Design". In *International Journal of Computer Systems - Science & Engineering*, Vol. 16 No. 5, Sep 2001, pp. 265-275.

[BF2000] O. Botti, V. De Florio, et al., "The TIRAN Approach to Reusing Software Implemented Fault Tolerance". In *Proc. of the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000)* (IEEE Comp. Soc. Press, Los Alamitos, CA), Rhodes, Greece, Jan. 19-21, 2000, pp. 325-332.

[BF99] O. Botti, V. De Florio, et al., "TIRAN: Flexible and Portable Fault Tolerance Solutions for Cost Effective Dependable Applications". In *Proc. 5th Int. Euro-Par Conference on Parallel Processing (EuroPar'99)*, P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, D. Ruiz (Eds.), Lecture Notes in Computer Science Vol. 1685 (Springer-Verlag, Berlin, Germany), Toulouse, France, Aug. 31-Sep. 3, 1999, pp. 1166-1170.

[BJ89] B.W. Johnson, "Design and Analysis of Fault -tolerant Digital Systems", Addison-Wesley Publishing Company Inc., Reading, M.A., USA,1989.

[BL91a] A. Burns, A. M. Lister, "A framework for building dependable systems", *The Computer Journal,* Vol. 34 No. 2, April 1991, pp. 73-181.

[BL91b] A. Burns, A M Lister, McDermid, "TARDIS: an architectural framework for timely and reliable distributed information systems". In *Proc. Sixth Australian Software Engineering Conf.* Sydney, Australia, July 1991, pp. 1-15.

[BL95] M. Barbacci, T. H. Longstaff, et al., "Quality Attributes", Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh Pennsylvania 15213 USA, December 1995.

[BR75] B. Randell, "System Structure for Software Fault Tolerance". In *IEEE Transactions on Software Engineering,* SE Vol. 1 No 2, 1975, pp. 220–232.

[BW94] A. Burns, A. J. Wellings, "HRT-HOOD: a structured design method for hard real-time systems", *Real-Time Systems Journal*, Vol.6 No.1, Jan. 1994, pp.73-114.

[BW95] A. Burns, A. Wellings, "HRT-HOOD: a structured design method for hard real-time Ada systems", Elsevier Science BV, 1995, ISBN 0-444-82164-3.

[CA78] L. Chen, A. Avizienis, "N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation". In *Proceedings of the 8th International Symposium on Fault–Tolerant Computing Systems (FTCS–8)*, Toulouse, France, 1978, pp. 3–9.

[CL2001] L.M. Cysneiros, J.C.S.P. Leite, et al., "A Framework for Integrating Non-Functional Requirements into Conceptual Models", *Requirements Engineering Journal*, Vol. 6, Issue 2, Apr. 2001, pp. 97-115.

[CL2004] L.M. Cysneiros, J.C.S.P. Leite, "Non-Functional Requirements: From Elicitation to Conceptual Model". In *IEEE Transactions on Software Engineering*, Vol. 30 No. 5, May 2004, pp.328-350.

[CN2000] L. Chung, B.A. Nixon, et al., "Non-functional Requirements in Software Engineering", Kluwer Academic Publishers, 2000.

[DB2000] G. Dondossola, O. Botti, "System fault tolerance specification: proposal of a method combining semi-formal and formal approaches". In *Proc. of Int. Conf. FASE2000, part of ETAPS2000 - The European Joint Conferences on Theory and Practice of Software,* Berlin, D, March 2000, LNCS, No. 1783, Springer-Verlag, Berlin, Heidelberg, New York, 2000, pp. 82-96.

[DepAuDE D1.1] D1.1: Collection of dependability requirements for embedded distributed automation systems in dynamic environments, DepAuDE Deliverable, July 2001.

[DepAuDE D1.4] D1.4: Dependability requirements in the development of wide-scale distributed automation systems: a methodological guidance, DepAuDE Deliverable, 2003.

[DepAuDE D2.1 & D2.2] D2.1 and D2.2: Updated Investigation, evaluation, and selection, DepAuDE Deliverable, 2002.

[DepAuDE D8.6] D8.6: Final Report, DepAuDE Deliverable, 2003.

[DepAuDE] DepAuDE project website, April 22, 2004, http://www.depaude.org/

[DF2001] G. Deconinck, V. De Florio, G. Dondossola, et al., "Distributed embedded automation systems: dynamic environments and dependability". In *Supplement of the Int. Conf. On Dependable Systems and Networks (DSN2001 - Special Track: European Dependability Initiative)*, Göteborg, Sweden, July 1-4 2001, pp. D16-D19.

[DF2002] G. Deconinck, V. De Florio, et al., "The EFTOS approach to dependability in embedded supercomputing". In *IEEE Transactions on Reliability*, Vol. 51, Mar. 2002, pp. 76–90.

[DSE] Dedicated Systems Encyclopedia, http://www.omimo.be/encyc/techno/terms/defini/def.htm

[DW98] D. D'Souza, A.C. Wills, "Objects, components, and frameworks with UML: The Catalysis Approach", Addison-Wesley: Reading, MA, USA, 1998.

[EC2000] European Dependability Initiative: Inventory of EC Funded Projects in the area of Dependability, Issue 2.2, 11 January 2000.

[FL92] C. J. Fidge, A. M. Lister, "A disciplined approach to real-time systems design", *Information and Software Technology*, Vol. 34 No. 9, September 1992, pp. 603-610.

[FL93] C. J. Fidge, A. M. Lister, "The challenges of non-functional computing requirements". In *Seventh Australian Software Engineering Conference (ASWEC'93)*, Sydney, September 1993, pp. 77-84.

[FTHH] The University of York, Department of Computer Science, Real-Time Systems Research Group, http://www.cs.york.ac.uk/rts/node12.html

[GD2001] G. Deconinck, Presentation on overview of DepAuDE at Pan-Dependability Workshop, Laboratory for Analysis and Architecture of Systems (LAAS), Toulouse, France, December 10-12, 2001.

[GG95] G. Gennaro, "Hierarchical Object Oriented Requirements Analysis", *Preparing for the Future*, Vol. 5 No. 3., September 1995.

[GM2002] J.-C. Geffroy, G. Motet, "Design of Dependable Computing Systems", Kluwer Academic Publishers, 2002.

[GN2000] A. Gokhale, B. Natarajan, et al., "DOORS: Towards high-performance fault-tolerant CORBA". In *Proc. 2nd Intl. Symp. Distributed Objects and Applications (DOA '00)*, Sept. 2000.

[GR2003] P. A. de C. Guerra, C. Rubira, et al., "Fault-Tolerant Software Architecture for Component-Based Systems". In *Lecture Notes in Computer Science,* Vol. 2677, Springer, 2003, pp. 129-149.

[HL74] J.J. Horning, H.C.Lauer, et al., "A Program Structure for Error Detection and Recovery". In E. Gelenbe and C. Kaiser (eds.), *Lecture Notes in Computer Science* **16**, Springer, 1974, pp. 171–187.

[HOORA] HOORA site, 2003, http://www.hoora.org/index.htm.

[HRM4] HOOD Reference Manual, Issue 4, 1995. Available at ftp://ftp.estec.esa.nl/pub/wm/wme/HOOD/HRM4.tar.gz.

[HS86] H. R. Simpson, ''The Mascot Method'', *Software Engineering Journal*, Vol. 1 No. 3, May 1986, pp. 103-120.

[JC2002] J. Jürgens, V. Cengarle, et al., "Critical Systems Development with UML", No. TUM-I 0208 in TUM technical report, 2002. UML'02 satellite workshop proceedings.

[JJ2003] J. Jürgens, "Developing safety-critical systems with UML". In *Proc. UML 2003 Conference*, LNCS 2863, Springer-Verlag 2003, pp. 360-372, San Francisco, California, USA.

[JJ2004] J. Jürgens, "Secure Systems Development with UML", Springer-Verlag, 2004 (to be published).

[JK2003] J. Kienzle, "Software Fault Tolerance: An Overview". In *Ada-Europe '2003*, 2003, Lecture Notes in Computer Science 2655, Springer-Verlag, pp. 45-67.

[JL92] J.-C. Laprie (Ed.), "Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese". In *Dependable Computing and Fault Tolerance*, Vol. 5, Springer-Verlag, Wien New York, 1992, 265 pages.

[JL96] J-C Laprie, "Software-based Critical Systems". In *15th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'96)*, Vienna, Austria, Springer, (1996), pp. 157-170.

[JL98] J.-C. Laprie, "Dependability of computer systems: from concepts to limits". In *Proc. of IFIP International Workshop on Dependable Computing and Its Applications (DCIA98)*, Johannesburg (South Africa), Jan. 12-14 1998.

[JR97] J-P Rosen, "HOOD An Industrial Approach for Software Design", 1997, HOOD User Group (Ed.), 232 pages, ISBN 2-9600151-0-X.

[KL2000] M. Kaâniche, J.-C. Laprie, et al., "A Dependability-Explicit Model for the Development of Computing Systems". In *19th International Conference on Computer Safety, Reliability and Security*

(SAFECOMP'2000), Rotterdam (Pays-Bas), 24-27 October 2000, Springer, ISBN 3-540-41186-0, pp. 107-116.

[KN99] L. Kabous, W. Nebel, "Modeling Hard Real Time Systems with UML The OOHARTS Approach". In *Proc. UML'99 Conference*, LNCS 1723, pp. 339-355, Springer-Verlag, 1999.

[KS98] K. Kim, C. Subburaman, "ROAFTS: A Middleware Architecture for Real-Time Object Oriented Adaptive Fault Tolerance Support". In *Proc. of the IEEE High Assurance Systems Engineering*, Nov 1998, pp. 50-57.

[LA90] P. A. Lee, T. Anderson, "Fault Tolerance - Principles and Practice". In *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, 2nd ed., 1990.

[LP2001] L.L. Pullum. "Software Fault Tolerance Techniques and Implementation", Artech House, Inc., 2001.

[LS2003] Y. Liu, P. Sinha, "A survey of generic architectures for dependable systems", IEEE Canadian Review, Spring, 2003.

[LT93] Loopback Testing, July 2004, http://www.hosenose.com/radio/support/loopback.htm

[MA92] M. J. Aslett, "An Overview of the HOOD Method". In *IEE Colloquium on Introduction to Software Design Methodologies*, Jun 1992, pp. 5/1-5/4.

[MAFTIA] MAFTIA project website, http://www.newcastle.research.ec.org/maftia/

[MB2001] R. Malan, D. Bredemeyer, "Defining Non-Functional Requirements", White Paper 2001, Bredmeyer Consulting.

[MC2003] M. Dal Cin, "Extending UML towards a useful OO-language for modeling dependability features". In the Ninth *IEEE Workshop on Object-Oriented Dependable Real-Time Systems*, October 2003.

[NISA95] "A Conceptual Framework for System Fault Tolerance", March 30 1995, Centre for High Integrity Software Systems Assurance, NIST.

[PA99] D. Powell, J. Arlat, *et al.*, "GUARDS: A generic upgradable architecture for real-time dependable systems". In *IEEE Trans. Parallel and Distributed Syst.*, Vol. 10, June 1999, pp. 580–597.

[PB93] P.A. Barrett, "Delta-4: An open architecture for dependable systems". In IEE Colloquium on *Safety Critical Distributed Systems*, 1993, pp. 2/1-2/7.

[PR92] P.J. Robinson, "Hierarchical Object-Oriented Design", Prentice Hall, 1992, ISBN 0-13-390816-X.

[REE98] NASA REE, Annual Report for FY'98, Jet Propulsion Laboratory, California Institute of Technology. Available at http://www-ree.jpl.nasa.gov/fy98_reports/rsft.html.

[RL2004] C. M. F. Rubira, R. de Lemos, et al., "Exception handling in the development of dependable component-based systems". In *Software – Practice and Experience*, 2004. To appear.

[RL95] B. Randell, J.-C. Laprie, et al., *ESPRIT Basic Research Series: Predictably Dependable Computing Systems*, Springer-Verlag, 1995.

[RR99] M. Rebaudengo, M. S. Reorda, et al., "Soft-error Detection through Software Fault-Tolerance techniques". In *Proc. of IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'99)*, Albuquerque, NM, USA, November 1-3, 1999, pp. 210-218.

[RX95] B. Randell, J. Xu, "The Evolution of the Recovery Block Concept", Chapter 1, pp. 1 – 21, in Lyu, M. R. (Ed.): Software Fault Tolerance, John Wiley & Sons, 1995.

[SN] Search Networking, 2004, www.searchnetworking.com.

[TD2001] R. Tirtea, G. Deconinck, "A Survey of Middleware and its Support for Fault Tolerance". In *Proc. 6th Int. Conf. Engineering of Modern Electric Systems (EMES-2001)*, Felix-Spa, Romania, May 24-26, 2001, 6 pages.

[TIRAN D1.1] D1.1 - Requirement specification V2, TIRAN Project Deliverable, October 1999, confidential.

[TIRAN D3.3] D3.3 – Guidelines for framework users V4, TIRAN Project Deliverable, October 2000, confidential.

[TIRAN D7.9] D7.9 – Project Final Report, TIRAN Project Deliverable, October 2000, confidential.

[UML2003] UML Revision Task Force. OMG UML Specification v. 1.5. OMG Document ad/03-03-01. Available at http://www.uml.org, 2003.

[VV2001] E. Verentziotis, T. Varvarigou, et al., "Fault tolerant supercomputing: a software approach". In *International Journal of Computer Research*, Vol. 10, No. 3, Nova Scotia Publishers Inc., 2001, pp. 401-413.

[WE72] W.R. Elmendorf, "Fault Tolerant Programming". In *Proceedings of the 2nd International Symposium on Fault–Tolerant Computing Systems (FTCS–2)*, Newton, MA, 1972, pp. 79–83.

Acronyms

DepAuDE	**Dep**endability for embedded **Au**tomation systems in **D**ynamic **E**nvironments
EFTOS	Embedded Fault-Tolerant Supercomputing
ESA	European Space Agency
ESPRIT	European Strategic Programme for Research in Information Technology
FT	Fault-tolerance
HOOD	Hierarchical Object-Oriented Design
HRT-HOOD	Hard Real-Time HOOD
NFR	Non-functional Requirement(s)
OMG	Object Management Group
OO	Object-Oriented
OOHARTS	Object-Oriented Hard Real-Time System
QoS	Quality-of-Service
RT	Real-Time
TARDIS	Timely and Reliable Distributed Information Systems
TIRAN	TaIlorable fault toleRANce frameworks for embedded applications
UML	Unified Modeling Language