A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

RELIABILITY-ORIENTED VERIFICATION OF MISSION-CRITICAL SOFTWARE SYSTEMS

ROBERTO PIETRANTUONO

Tesi di Dottorato di Ricerca
(XXII Ciclo)
Novembre 2009

Il Tutore
Prof. Stefano Russo

Il Coordinatore del Dottorato
Prof. Francesco Garofalo

Dipartimento di Informatica e Sistemistica

RELIABILITY-ORIENTED VERIFICATION OF MISSION-CRITICAL SOFTWARE SYSTEMS

By
Roberto Pietrantuono

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
“FEDERICO II” UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 – NAPOLI, ITALY
DECEMBER 2009

Table of Contents

List of Tables

List of Figures

Introduction

A *mission-critical* system is a system whose failure or disruption may lead to catastrophic loss in terms of cost, damage to the environment, or even human life. Mission-critical systems are adopted in a growing number of areas, ranging from banking to e-commerce, from avionics to railway, from automotive to health care scenarios. In such contexts, high level of reliability represents a crucial requirement to satisfy. However, activities needed to achieve the desired reliability can require huge development cost. In particular, it has been showed that the greatest part of the total development and maintenance cost is due to the *verification process* [1], especially in large systems.

This is mainly due to the inadequateness of current verification strategies (i) to deal with complexity and size of today's software systems, and (ii) to sufficiently cope with issues related to *high-reliability* demands (e.g., few strategies are specifically conceived to improve reliability).

In several critical application domains the reliability level to be achieved (along with other quality goals) is imposed by domain-specific standards, such as CENELEC [2], in the field of railway applications, IEC 61508 [3], for electrical/electronic systems, or the DO-178B [4] for ATC systems. In order to achieve the required reliability, these standards also suggest potential techniques that can be adopted along the development cycle, but neglecting cost and effectiveness issues. The guidelines they provide are quite general, as also noticed by [5], since their purpose is not to define what techniques a company must use or what is their impact on company's cost. Hence, there is a gap between what they suggest and strategies that can actually be adopted.

This poses serious difficulties to companies, which on one hand are constrained to meet predefined reliability goal, whereas, on the other hand, are required to deliver systems at acceptable cost and time. This thesis focuses on verification activities mainly impacting the reliability-cost trade-off. It aims to provide means for an effective reliability-oriented verification in large critical software systems.

Currently, reliability is rarely adopted as pilot criterion to drive verification activities, since it is difficult to quantitatively evaluate the impact of crucial choices in the verification process (such as *what techniques* should be adopted) on the final reliability. Most often such choices are left to the engineers' intuition, which base their decisions on personal expertise and on past experience. However, we claim that in systems where reliability is the main concern, activities in the verification phase *should systematically be focused on heighten the final reliability through the adoption of objective and quantitative criteria.*

Moreover, current verification processes often underestimate some phenomena which effects are visible only during the operational phase. One of this is the software aging phenomenon [6], in which progressively accrued error conditions lead to a gradual degradation and to the system failure after some period. Even though software aging is not easily detectable at testing time, in highly reliable systems it cannot be neglected, as demonstrated by the catastrophic accidents that it caused in the past [7].

Thesis Contributions

Focus of thesis is on the effective *reliability-driven* verification of large critical software systems. Many challenges have to be addressed towards this aim. These range from the *accurate identification* of the most critical software components for effective effort allocation, to the *selection of most proper verification technique(s)* to adopt for a software component in hands, up to the adoption of *suitable techniques* able to detect faults that typically manifest only at runtime (e.g., software aging).

Indeed, in order to build effective plans, efforts for verification should be conveniently allocated to the various parts of the system, e.g., devoting greatest resources to the parts mostly affecting the quality objective. This requires engineers to correctly *identify the most critical components/subsystems* in the software architecture from reliability's point of view. However, identifying parts of a complex system that are the major contributors to its unreliability is not an easy task, and consequently the testing resource allocation is most often based on engineers' judgement. Sometimes, engineers tend to judge as "most critical" components that are the most complex ones, or those that are the most used ones and devote most of the testing efforts to them, making wrong allocations and hence wasting precious resources.

The first step towards effective verification planning addressed by this work is therefore the convenient allocation of resources available for verification. It requires the adoption of allocation policies conceived to best distribute the efforts (that may be intended as man/month, calendar time, number of test cases, CPU time, etc.) among the various parts of the system, considering the reliability objective to satisfy.

Even distributing the resources correctly, a critical choice regards the verification techniques to adopt in the plan. In the context of highly reliable systems, the impact of different kinds of verification techniques on the final reliability should be considered to drive the selection. However, the relationships between techniques and reliability that they are able to deliver are currently not known, since most of studies in the literature neglect how a given technique impact reliability[1]. Moreover, a reliability-oriented techniques selection should also consider the influence of the system being tested: two distinct applications of the same technique to distinct software systems yield, in general, different results. Thus engineers have also to understand how choosing a proper combination that best adapts to the software being developed. This choice is often based, again, on engineers intuition and experience. *Hence, the second relevant issue addressed in this work is how to combine techniques in order to design a verification plan oriented to improve the reliability attribute and tailored for the specific system being developed.*

To address the above mentioned issues, the dissertation proposes a solution for the improvement of verification activity effectiveness oriented to deliver highly reliable software systems, where the most critical choices are supported by means of quantitative criteria. In particular, the proposed solution addresses the outlined challenges by:

- **Defining a policy for cost-effective allocation of verification resources.** The study explores the existing solutions for modelling the efforts distribution and proposes a way to allocate resources to allow each subsystem to achieve the necessary reliability to meet the overall system reliability level.

- **Improving the effectiveness of reliability-driven verification techniques selection.** There is no technique that is the best one for all contexts; the choice of techniques should be based on the specific context where it will be employed. An empirical approach is proposed to address the issue of how to select and to apply the proper verification techniques for a software system. It aims (i) at analyzing, by running a controlled experimental campaign, the effectiveness of various verification techniques in detecting faults, in improving reliability and with respect to different

[1] Note that detecting more faults does not imply improving the reliability. The reliability increase requires the removal of those faults which occurrence at operational time is more frequent

types of faults that potentially affect the system; and (ii) at characterizing a system by the tester's point of view, by investigating (and inferring) the potential relationship between the types and the number of faults present in a software system, and some relevant software features, expressed by means of common software metrics. This would ease a *reliability-driven system-specific* techniques selection.

- **Taking into account the software aging phenomenon for planning verification.** In particular, this work intends to consider the aging phenomenon, typically taken into account at runtime, by estimating aging proneness of the software being developed in order to take proper actions already in the testing phase. To the best of author's knowledge, this is the first attempt to relate the software aging phenomenon to the features of the software being tested.

The work also defines a procedure to exploit at their best results of the outlined contributions in order to let testers determine the most suited set of techniques, their best order of application and the final confidence at which reliability is improved. The procedure is conceived to iteratively refine results across the developed projects.

The dissertation is organized as follows:

Chapter 1 provides the needed background on software faults, on software reliability analysis and on software verification strategies.

Chapter 2 deeply discusses the existing related work, analyzing the state-of-the-art concerning resources allocation, verification techniques and their relationship with reliability, and software aging phenomenon. It then gives a synthetic description of the contribution provided by this work, which is then exploded in the successive chapters.

Chapter 3 presents the proposed approach to cope with the careful allocation of resources available for verification to the various parts of a system. It describes the adopted model, the information it needs and the experimental results.

Chapter 4 presents an empirical analysis of verification techniques revealing how verification techniques behave in various conditions and what is their impact on the reliability attribute. Techniques are evaluated in their effectiveness in detecting faults, in improving reliability, in their cost and most importantly with respect to different types of faults.

In chapter 5, a further empirical analysis aiming at inferring what relation stands between fault types and some relevant software features is presented. Results of this analysis complement the analysis conducted in chapter 4, in that it gives the tester the opportunity to bind verification techniques to the software features, through the knowledge of potential fault types affecting the software. Software aging impact is also discussed in this chapter.

Finally, chapter 6 describes the steps that a quality manager should take in order to effectively plan verification activities. The chapter describes how such steps should be incorporated in a verification process in a self-refining way; i.e., in order that results of each process application iteratively enrich the empirical data on which the procedure is based and improve the final accuracy.

Chapter 1

Software Reliability and Verification

Pervasiveness of software in our life induces many people to rely on computing systems and on services they offer. People are aware that computing systems can support, and in some cases replace, their activities in many contexts. However, this is not sufficient alone to make people reasonably trust software systems. In many cases, systems are not only required to deliver the intended service; but also that they are always able, under well-defined circumstances, to deliver the intended service within a reasonable confidence level. But in the real world software systems, as any other kind of system, not always succeed in doing what they are required; in certain conditions, system's behaviour deviates from the expected one, and cause damages often unforeseeable and catastrophic. Such deviations are caused by "faults" in the system, which differently from other engineering disciplines, are very hard to detect and remove. This chapter first discusses the concept of software fault, explaining how it can lead a system to undesired behaviours. In particular, faults are explored and classified with respect to their typology and reproducibility. The effects of fault activation on reliability provided by the system is then discussed. The latter part is devoted to the set of activities typically undertaken by engineers to uncover faults and deliver highly reliable systems: the software verification. Basic background on verification process and several verification techniques are briefly surveyed in this section.

1.1 Software Faults

1.1.1 Fault-error-failure

The function of a system is what the system is intended to do and is described by the functional specification in terms of functionality and performance [8]. The service delivered by a system is what the system actually does to implement its function. When a system is not able to deliver the correct service (i.e., to implement its function), a service failure, or simply a failure, is said to be occurred. Since a service is a sequence of the system's external states (i.e., states perceivable by the user), a failure means that at least one of such states deviates from the correct one. The deviation is called an error. The adjudged or hypothesized cause of an error is called a ***fault***. Referring to the well-known and widely accepted taxonomy presented in [8], faults, errors and failures are related as shown in Figure 1.1.

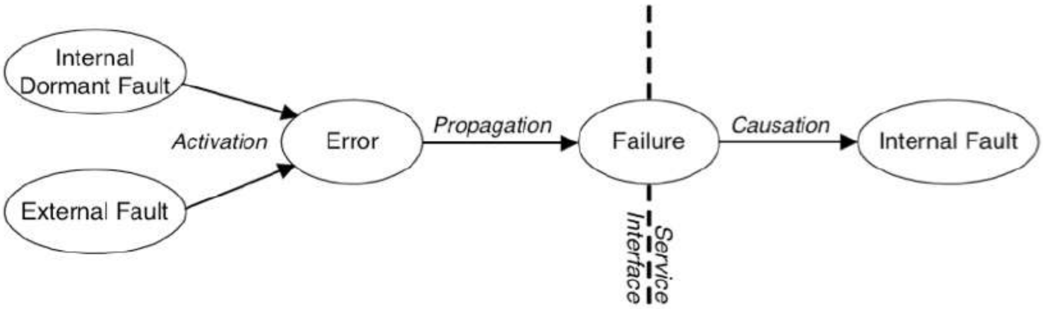


Figure 1.1: Fault-Error-Failure Chain

A fault is said to be active when it produces an error; otherwise it is dormant. If the produced error propagates up to the service interface a failure occurs. An error which does not lead the system to failure is said to be a latent error. A failure of a system component causes an internal fault of the system that contains such a component, or causes an external fault for other system(s) that receive service from it. The ability to identify the activation pattern of a fault is the fault activation reproducibility. Among the many classes of faults identified in [8], the ones attributable to software are 13, depicted in Figure 1.2.

Even though there are many other types of faults, software faults are the main source of failures in today's systems [9], [10]. Hardware fault tolerance, fault avoidance and management, reliability/availability modelling are relatively well developed with respect to software; as a consequence, system outages are more often due to software faults.

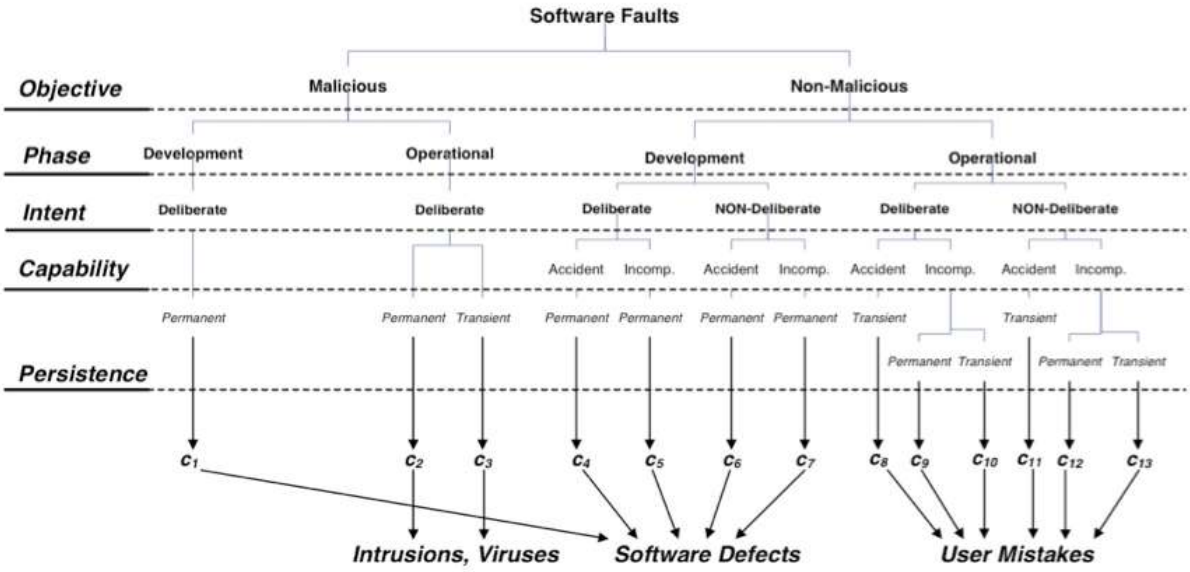


Figure 1.2: Classification od Software faults

Laprie et al. classified software faults according to several dimensions, such as the phase of creation or occurrence (development or operational), the objective (malicious or non-malicious), the intent (deliberate or not), its capability (accidental or incompetence), and its persistence (permanent or transient).

Faults introduced during operational phase can be either permanent or transient and are due to user mistakes or attacks. Faults introduced in the development stage are always permanent, and are commonly known as "software defects". This thesis focuses on this class of faults.

The activation of software faults can cause the system to fail, if it reaches the service interface. This can occur during system verification (hopefully), during operational phase, or never (in case of dormant faults). The objective of verification is to cause these faults to be activated, in order to remove them prior to operational phase. The software fault types reported in Figure 1.2 can be classified adopting various criteria, depending on what characteristics we want to emphasize. In the following, they will be classified according to their semantics, adopting the *Orthogonal Defect Classification* (ODC) scheme, and according to their reproducibility, distinguishing faults that exhibit a deterministic behaviour from others that, instead, seem to be non-deterministic.

In the rest of this section, the term “software fault”, or simply “fault” will refer exclusively to faults introduced in the development phase (i.e., software defects), either malicious or not.

1.1.2 Fault Types: the Orthogonal Defect Classification

Orthogonal Defect Classification (ODC) is a scheme to capture the semantics of software defects that was first presented in 1992 [11]. Its main goal is the definition of defect attributes that enable in-process feedback to engineers by extracting key information on the development process from defects. Classification and subsequent analysis of ODC data allow engineers to evaluate various phases of the software life cycle (design, development, test and service) and the maturity of the product.

ODC overcomes the limitation of other defects/process analysis techniques, such as the *Root*

Cause Analysis (RCA), and *statistical growth modelling.* Indeed, RCA provides enough details on each defect, but it requires substantial investment and it is based on qualitative reasoning, not lending itself to measurement and quantitative analysis. Growth modelling, on the other hand, provides an easy way to monitor trends, but it is not capable of suggesting corrective actions due to the inadequate capture of the semantics behind the defects. ODC lies between these two extremes, since it attempts to preserve the clearness and easiness of qualitative approaches and the measurability of quantitative analyses. Its wide spreading undoubtedly confirms ODC as a fundamental milestone in the analysis of the dependability of software systems.

The ODC approach follows two main steps: fault classification and fault analysis.

Classification is done in two phases: when faults are detected and when they are fixed. At detection time, key information is recorded about: the activity executed when the fault is revealed, the trigger that exposed it, the perceived or actual impact of the fault on the customer. Trigger and activities are related. A defect trigger is a condition that allows a defect to be activated. Examples of activities are *Design Review* and *Code Inspection*, in which triggers may be the *checking for design conformance* or *backward compatibility*, or*System Test*, in which a trigger may be the *workload testing session.*

The most used defect trigger categories are:

- **Boundary Conditions** - Software defects were triggered when the systems ran in particularly critical conditions (e.g., low memory).
- **Bug Fix** - The defect surfaced only after another defect was corrected: this happens

either because the first defects was covering the second one, or because the fix was not successful, introducing a new bug.

- **Recovery** - The defect surfaced after the system recovered from a previous failure.
- **Exception Handling** - The defect surfaced after an unforeseen exception-handling path was executed.
- **Timing** - The defect emerged when particular timing conditions occurred.
- **Workload** - The defect surfaced only when particular workload condition occurred (e.g., only after the number of concurrent requests to serve was higher than a given threshold).

At **fix time** the developer has to record: the target (i.e., the entity fixed to remove the fault), the source (i.e., the origin of the faulty modules, such as "in-house", library, outsourced or ported from other platforms), the age of the faulty element (new, old, rewritten, or re-fixed code), and its type. The defect type is the kind of information that may be trickiest to classify: hence, the ODC defines types that are as much simple as possible and orthogonal, in order to avoid confusion. The idea is that it should be quite obvious for programmer to classify them. In each case a distinction is made between something missing and something incorrect. In [11] the following defect types have been defined.

- **Function** - The fault affects significant capability, end-user interfaces, interface with hardware architecture or global data structures and should require a formal design change. Usually these faults affect a considerable amount of code and refer to capabilities either implemented incorrectly or not implemented at all.

- **Interface** - This defect type corresponds to errors in interacting with other components, modules or device drivers, via macros, call statements, control blocks or parameters list.

- **Assignment** - The fault involves a few lines of code, such as the initialization of control blocks or data structures. The assignment may be either missing or wrongly implemented.

- **Checking** - This defect addresses program logic that has failed to properly validate data and values before they are used. Examples are missing or incorrect validation of parameters or data in conditional statements.

- **Timing/Serialization** - Missing or incorrect serialization of shared resources, wrong resources serialized or wrong serialization technique employed. Examples are deadlocks or missed deadline in hard real time systems.

- **Algorithm** - This defect includes efficiency and correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.

- **Build/package/merge** - Describe errors that occur due to mistakes in library systems, management of changes, or version control. Rather than being related to the product under development, this defect type is mainly related to the development process, since it affects tools used for software development such as code versioning systems.

- **Documentation** - This defect type affects both publication and maintenance notes. It has a significant meaning only in the early stages of software life cycle.

The detailed information on faults allows for numerous subsequent analyses. Examples of what can be done are:

- **Distribution of Fault Types vs. activities**: for instance, algorithmic faults are mainly targeted by unit testing, and a high proportion is expected to be found; if this does not happen, then the unit tests may be not well designed.
- **Distribution of Trigger over time during field test**: faults corresponding to simple activation conditions should arise early during field test, while complex activation faults should appear later. In both cases, the detection rate should asymptotically decrease. Unexpected distributions of triggers over time may indicate poor system or acceptance test.
- **Age Distribution over target code**: most faults should be located in new and rewritten code, whereas few faults should be found in old or re-fixed code, which has already been tested. Different patterns may reveal holes in the fault tracking process.

ODC is used in this work as fault classification scheme in order to evaluate and consider the impact of distinct defect types on the selection of the most suited verification techniques.

Extension to the ODC

In 2003, the ODC has been extended by Madeira and Duraes [12], [13], which refined it by taking into account other factors, such as those related to language programming constructs being used. The original ODC relates faults to the way they are corrected: since the same fault can be usually corrected in different ways, Madeira and Duares considered for each ODC category a further classification in order to improve the accuracy of description.

In fault types classification, it may happen that several different faults are classified in the same category, even if their nature and their activation path is totally different. According to these authors, a fault in any ODC class can be further characterized by a programming language construct that can be either missing, wrong or superfluous. Developers may come to this detailed classification by adopting the ODC as a first step, and then, in a second step, by grouping faults according to the nature of the defect, defined from the mentioned “programming constructs” perspective.

Finally, in the third and last step, faults are further refined and classified in specific types. As an example of such fault types, it is possible to consider Missing function calls (MFC), which is a particular kind of algorithm fault in which a required function call is missing. Field data collected in [13] reported more than 20% of faults belong to this category. Distribution of occurrence of any ODC fault categories is also reported in the same work [13].

1.1.3 Fault Activation: Bohrbugs, Heisenbugs, Mandelbugs

The ODC classifies software faults according to their semantic, i.e., categorizing defect types by looking at the semantic of their fix. This classification is very useful from the process improvement perspective, in that it can provide insights about deficiencies of process development stages.

A crucial concern for software engineers is to understand how a fault, of any type, may be activated and how it can be reproduced. Software fault reproducibility was first discussed in 1986 [14], by Jim Gray. In this work, Gray distinguished faults whose activation is easily reproducible (e.g., through a debugger), i.e., solid or hard faults, from faults whose activation is not systematically reproducible, i.e, elusive or soft faults.

Solid faults manifest consistently under a well-defined set of conditions and that can easily be isolated, since their activations and error propagations are relatively simple. Soft faults, instead, are intricate enough that their activation conditions depend on complex combinations of the internal state and the external environment. The conditions which activate the fault occur very rarely and can be very difficult to reproduce.

In Gray's paper, the first class of faults (i.e., solid) were named **"Bohrbugs"**, recalling the physicist Niels Bohr and his rather simple atomic model: *"Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring"*. Thus a Bohrbug does not disappear or alter its characteristics when it is activated. These include the easiest bugs to fix (where the nature of the problem is obvious), but also bugs that are hard to find and to fix, which remain in the software during the operational phase. A software system with a Bohrbug is analogous to a faulty deterministic finite state machine.

Soft faults were instead defined as those faults for which *"if the program state is reinitialized, and the failed operation is retried, the operation will not fail a second time"*. Such kind of faults were named **"Heisenbugs"**, referring to the physicist Werner Heisenberg and his Uncertainty Principle.

Based on Gray's paper, researchers have often equated Heisenbugs with soft faults. However, when Bruce Lindsay originally coined the term in the 1960s (while working with Jim Gray), he had a more narrow definition in mind. In his definition, Heisenbugs were envisioned as *"bugs in which clearly the system behaviour is incorrect, and when you try to look to see why it's incorrect, the problem goes away"*. In this sense the term recalls the uncertainty principle, in that the measurement process (in this case the fault probing) disturbs the phenomenon to be measured (the fault).

A software system with a Heisenbug is analogous to a faulty non-deterministic finite state machine. One common example is a bug that occurs in a release-mode compile of a program, but not when researched under debug-mode; another is a bug caused by a race condition. One common reason for heisenbug-like behaviour is that executing a program in debug mode often cleans memory before the program starts, and forces variables onto stack locations, instead of keeping them in registers. Another reason is that debuggers commonly provide watches or other user interfaces that cause code to be executed, which can, in turn, change the state of the program. Moreover, many Heisenbugs are caused by uninitialized variables. Hence, Heisenbugs are, for their nature, very difficult (if not impossible) to reproduce, due to their non-deterministic activation.

However, between Bohrbugs and Heisenbugs behaviour there is another class of bugs that has been identified later, which cannot be classified in neither of the two categories. They are, like Heisenbugs, very hardly reproducible; but their activation is just apparently non-deterministic, i.e., they are deterministically activated by a particular exact condition (like Bohrbugs), but detecting this condition is so difficult that the bug can be considered as non-deterministic.

In scientific literature these software defects are named **Mandelbugs** (which name derives from the name of fractal innovator Benoit Mandelbrot). According to this classification,

- *A Heisenbug is a bug that disappears or alters its characteristics when it is probed.*
- *A Mandelbug is a bug whose causes are so complex that its behaviour appears to be chaotic (but they are deterministic).*

However, currently there is no agreement in the literature and the term “Heisenbug” is used inconsistently: some authors accepts this classification; others use the term Mandelbugs as a synonym for Heisenbugs, since they claim that there is no way to distinguish a bug whose behaviour appears chaotic and a bug whose behaviour is actually chaotic.

On the other hand, Trivedi *et al.* [15] claim that Heisenbugs are actually a kind of Mandelbugs. He identified two main sources of complexity characterizing the occurrence of a Mandelbugs. The first one is that there may be a long delay between the fault activation and the final failure occurrence (e.g., because several error state are traversed before the failure or because the fault progressively accrues an abnormal condition until the system fails). This usually happens with complex software systems employing one or more Off-The-Shelf (OTS) items. The second source of complexity is due to the system-internal environment: fault activation and/or error propagation depend on interactions between conditions occurring inside the application and conditions that accrue within the system-internal environment (e.g., a fault causing failures due to side-effects of other applications in the same system, or a race condition caused by inadequate synchronization in multi-threaded software). According to this view, *"since the system-internal environment induces the change in [fault] behaviour, Lindsay's Heisenbugs are actually a type of Mandelbug"*. Referring to this classification:

- *Mandelbugs include those faults which activation is chaotic or non-deterministic (including Heisenbugs);*
- *Heisenbugs are a special case of Mandelbugs in which system-internal environment influences fault's behaviour in a specific application (hence causing the bug to "disappear" when probed).*

In the following we will refer to this classification. Due to their different nature, removing Bohr- or Mandel- bugs implies distinct techniques to be adopted. Indeed Bohrbugs, due to their simple activation conditions are more prone to be detected and fixed during the software design and verification phase. For instance, structured design, design review, formal analysis, unit testing, inspection, integration, system and acceptance testing are techniques that are employed in the development phase and that usually fix the greatest part of Bohrbugs.

On the other hand the residual bugs are rare cases, typically related to specific environmental conditions, limit conditions (e.g., out of storage, out of memory, buffer overflows, etc.) or race conditions, which may require a long time to be activated and to manifest; these are what we called Mandelbugs.

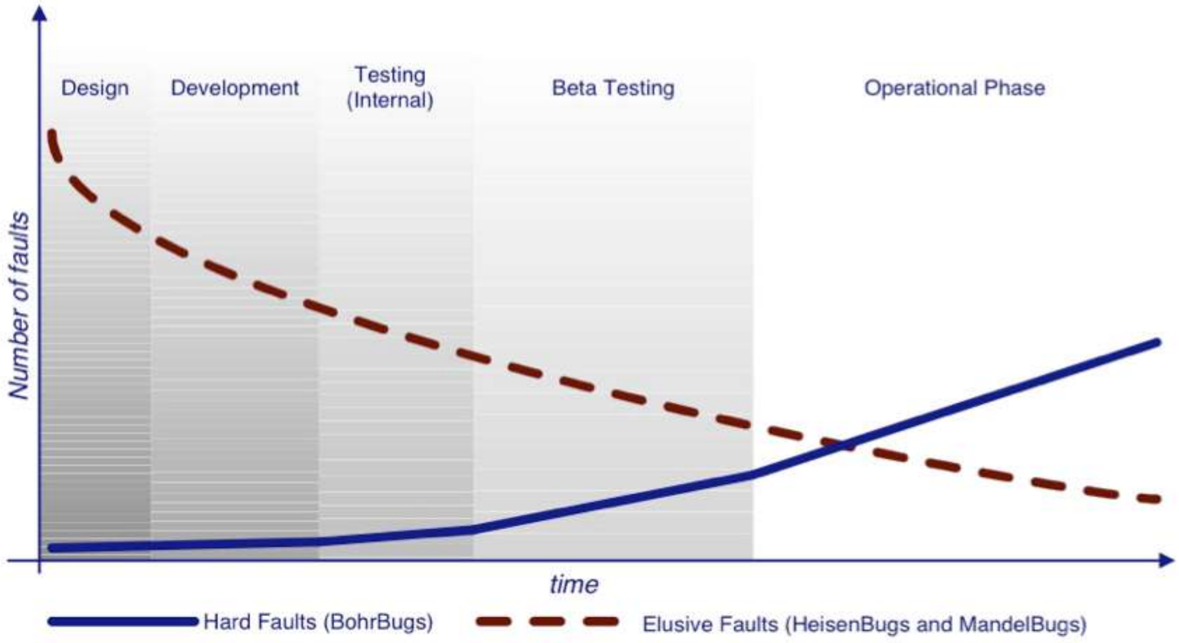


Figure 1.3: Evolution over time and software life cycle phase of reproducible and non-reproducible software faults

For these reasons, the number of detected Bohrbugs decreases over life cycle time (as depicted in Figure 1.3, by [16]) , becoming negligible after a long period of production. On the other hand, the number of Mandelbugs increases over time: during the development phase this number is very low, since the system under test runs always in the same environment and intricate activation conditions rarely occur; during the beta testing, when the

system is delivered out of the production environment, a consistent number of Mandelbugs are reported. In that stage, the system runs in several environments under workloads very different from the ones applied in the testing phase. This number further increases once the system is brought to the operational phase.
Given their nature, Mandelbugs are typically treated with the following recovery techniques [15], which aim to change the execution environment state: **"Micro-reboot" of individual component, Application restart, System reboot, Failover to a standby component (replicate)**.

One more category of bugs, which recently is gaining attention, are the so-called **"Aging-related bugs"**. Often, software systems running continuously for a long time tend to show a degraded performance and an increased failure occurrence rate. This phenomenon is usually called Software Aging [6]. There are several examples in the literature reporting real system failures due to software aging. It is typically caused by accrued error conditions, such as round-off errors, data corruption or unreleased physical memory. The preventive technique against this phenomenon is known as Software Rejuvenation, which consists in cleaning the system-internal environment without removing the bug in order to reset the accumulated error.
A typical example of Aging-Related bug is an unreleased memory region inside a program's heap area, i.e., memory allocated and never released. Also in this case, there is no complete agreement on the classification of these bugs: some authors relegate Aging-related bugs only in the Mandelbugs category, while others view these bugs as an intersection between

Mandelbugs and Bohrbugs classes. Indeed, Aging-related bugs can be either deterministic (e.g., a missing delete statement) or non-deterministic (e.g., a fault which is dependent on the message arrival order). However both kinds of bug require a long time to manifest (the aging phenomenon notion requires a delay between fault activation and failure occurrence). Hence, also deterministic Aging-related bugs will very likely manifest at the operational time (like Mandelbugs), being very difficult to note their effect at testing time. For this reason, in accordance with the adopted classification for Bohrbugs-Mandelbugs, we adopt this second solution (Aging bugs as subset of Mandelbugs), suggested by [15], for placing aging-related bugs in the classification scheme.

Figure 1.4 (by [15])summarizes the adopted bugs classification scheme and potential recovery techniques.

With respect to the classification scheme in Figure 1.4, this thesis deals with all the mentioned classes o faults. In fact, the focus of this thesis is on reliability-oriented software verification. Software verification clearly aims to remove as much Bohrbugs as possible in the least time. On the other hand, software reliability is a quality property related to the operational phase, because it is a user-perceived attribute. This means that it depends not only on the number of residual faults in the software, but also on their activation pattern. A verification oriented to achieve high reliability should also tend to remove those faults which activation is rare and that may occur during operation (i.e., Mandelbugs), trying to "anticipate" their occurrence during testing. The thesis copes with this issue to pursue its goal.

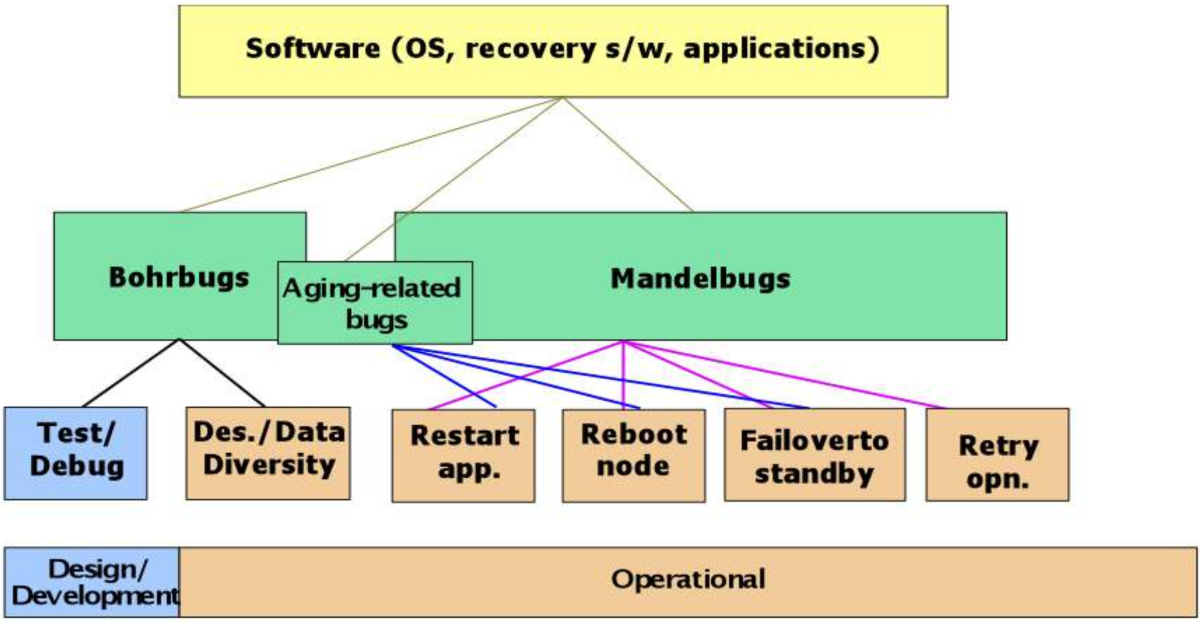


Figure 1.4: Fault Classification according to their Reproducibility

1.2 Software Reliability

1.2.1 Basic Notions

Reliability is a measure of continuous delivery of correct service. It is one of the attributes of dependability concept [8] that is particularly relevant for critical systems, where no down time can be tolerated. Differently from availability, which focuses on failures on a given instant of time, reliability emphasizes the occurrence of undesirable events in a specified time interval, often called Mission Time. A lowly available system may be highly reliable in a given mission time, depending on when the failures occur (e.g., a system may fail many times in one year, but never in the time interval that is crucial for system's mission, i.e., it fails always in times when failures are tolerable).

Reliability is often specified as a numeric value indicating a probability (typically, the probability that the system fails in a give time interval), or by means of its mean time to failure (MTTF), or its failure rate (that is the frequency with which a system fails). However, in a more formal way, reliability is a function of time, denoted by R(t): **it represents the probability that the system survives (does not fail) throughout [0,t]**. This function has the following properties:

1. The system is assumed to be working properly at time t = 0, i.e., R(0) = 1
2. The system cannot work forever without failure: $\lim_{t\to\infty}$ R(t) = 0
3. R(t) is a monotone decreasing function of *t*.

Let us call X a random variable denoting the time to failure of a system, and $F(t)$ the unreliability function, i.e., the cumulative distribution function of X. The reliability function can thus be written as:

$$R(t) = Pr(X > t) = 1 - F(t) \tag{1.1}$$

Reliability function can be approximated as "probability of survival" [17], using the frequency interpretation of probability. Consider a number of identical components, N_0, under test (or equivalently N_0 test cases of a system test session). Let us denote with $N_{f(t)}$ the number of components (or test cases) failed at time t and with $N_{s(t)}$ the number of survived components (or negative-outcome test cases, i.e., not failed executions), so that: $N_{f(t)} + N_{f(t)} = N_0$. Hence:

$$Pr(survival) = N_{s(t)}/N_0 = 1 - N_{f(t)}/N_0 \approx R(t) \tag{1.2}$$

In the limit as $N_0 \rightarrow \infty$, Pr(survival) is expected to approach to R(t) ($N_{s(t)}$ gets smaller and R(t) decreases). As shown in [17], taking derivatives on both sides, we have:

$$R'(t) = -(1/N_0) * N^{'}_{f(t)} \tag{1.3}$$

where $N^{'}_{f(t)}$ is the rate at which components fail; thus the ratio of equation 1.3 represents the failure density function of the variable X:

$$R^{'}_{(t)} = -f(t) \tag{1.4}$$

In this expression, f(t) Δt is the unconditional probability that a component will fail in (t, t+ Δt); the probability that component will fail in (t, t+ Δt) given that it did not fail until t is expected to be always greater than f(t)Δt . This conditional probability is:

$$GY(x|t) = P(t < X \leq t + x)/P(X > t) = (F(t + x) - F(t))/R(t) \tag{1.5}$$

Dividing this probability by the interval of time x and taking the limit, we obtain the instantaneous failure rate h(t):

$$h(t) = \lim_{x \rightarrow 0}(F(t + x) - F(t))/xR(t) = f(t)/R(t) \tag{1.6}$$

Thus, h(t)Δt represents the conditional probability that a component having survived till t will fail in (t, t + Δt]. By integrating both sides of equation 1.6, and using equation

1.4, the final reliability expression is obtained:

$$\begin{gathered}
\int_0^t h(x)dx = \int_0^t f(x)/R(x)dx \\
\int_0^t h(x)dx = \int_0^t -R'(x)(R(x)dx \\
\int_0^t h(x)dx = -\int_{R[0]}^{R[t]} dR/R \\
\int_0^t h(x)dx = -lnR(t) \\
R(t) = exp[-\int_0^t h(x)dx]
\end{gathered} \tag{1.7}$$

The instantaneous failure rate is also known as hazard rate, conditional failure rate or simply failure rate. Since in practice it is a small value, it is commonly expressed in failures over 10000 hours or in FIT (failure in time), i.e., failures per 10^9 or a billion hours. The cumulative hazard rate, H(t) = $\int_0^t h(x)dx$ is referred to as cumulative hazard. Note that if the time to failure (TTF) F(t) has an exponential distribution, the hazard rate is constant:

$$h(t) = f(t)/R(t) = \lambda e^{-\lambda t}/e^{-\lambda t} = \lambda \tag{1.8}$$

This means that the lifetime is independent of component age t (note that the exponential distribution is the only distribution with a constant failure rate), i.e., a component does not age stochastically and the probability of failure over an additional period δt is the same regardless of its current age. Conversely, a survival probability can decrease over time (due to aging), in which case the corresponding distribution function F(t) is an increasing failure rate (IFR) distribution, or can increase over time (e.g., if aging is beneficial for component lifetime), in which case the distribution is a decreasing failure rate (DFR) function. Usually, the failure rate of a system is not constant during all the system life-time, but it follows the so-called bath-tube form (empirically observed), i.e., a system experiences a decreasing failure rate when it is firstly deployed, due to infant-mortality failures, then it follows a rather constant failure rate during the operational life (that justifies the adoption of an exponential distribution for the useful-life phase), and, finally, it experiences an increasing failure rate at the end of its life, due to wear-out failures.

Common distribution for h(t) are the exponential, the hyper-exponential, the lognormal, and the weibull.

The **exponential distribution** was firstly adopted to model the time to failure and time to repair of electronic components. However, due to its simplistic memoryless property, the exponential distribution is not able to fit real data when multiple and dependent failure causes are involved.

If a process consists of alternate phases, that is, during any single experiment, the process experiences one and only one of many alternate phases, and these phases have exponential distributions, then the overall distribution is **hyper-exponential** [17]. This failure time distribution can be used to model failures which are the manifestation of different, independent and alternate underlying causes.

Recently, the **lognormal distribution** has been recognized as a proper distribution for software failure rates [18]. Many successful analytical models of software behaviour share assumptions that suggest that the distribution of software event rates will asymptotically approach lognormal.

The **weibull distribution** has been used to describe fatigue failures and electronic components failures. At present, it is perhaps the most widely used parametric family of failure distributions. The reason is that by a proper choice of its shape parameter, an increasing, a decreasing, or a constant failure rate distribution can be obtained. Therefore, it can be used for all the phases of the bath-tube mortality curve [17].

1.2.2 Reliability Modelling

Reliability can be evaluated by using several approaches, generally classified into two categories: *model-based* and *measurements-based*. Model-based approaches are widely used for reliability evaluation of complex software/hardware systems. They are based on the construction of a model that is a “convenient” abstraction of the system, with enough level of detail to represent the aspects of interest for the evaluation. The degree of accuracy of a model depends on the ability of the associated formalism to extrapolate the system features.

Models allow to suitably analyse a system architecture, to evaluate different configurations, to pinpoint performance/reliability bottlenecks, to make predictions and to compare design alternatives without physical implementation. They can be object of analysis or simulation. Analytic models are classified in combinatorial and state-based models. Models in the first category represent the structure of the system in terms of logical connection of working (failed) components in order to obtain the system success (failure). State space based models represent the behaviour of the system in terms of reachable states and possible state transitions.

Models can be solved analytically in closed-form or numerically depending on their computational complexity. When analytical solution is not available (or it is computationally expensive) simulation represents a viable alternative.

COMBINATORIAL MODELS

Combinatorial models have a simple and intuitive notation. They are easy to be designed and manipulated, and they can be efficiently analysed by means of combinatorial techniques. The system is typically divided into a set of non-overlapping modules, each one associated with either a probability of working, P_i (or a probability as function of time, e.g., R_i(t)). The goal is to derive the overall P_{sys} value (or function R_{sys}(t)), representing the probability that the system survives (until t). These models typically enumerate all the system states, by using combinatorial counting techniques to simplify the process.

Despite of their advantages (simple and intuitive analysis), combinatorial models suffer from a limited modelling power, mainly due to the assumptions they make:*(i) module failures are independent (i.e., statistical independence of events), (ii) once a module has failed, it is always assumed to yield incorrect results, (iii) once system enters a failed state, other failures cannot resume the system to functional state.*

Examples of combinatorial models are Reliability Block Diagrams (RBD) [19], [20] and Fault Trees (FT) [21].

RBDs use logical blocks to link a complex system state to the states of its components. A block, representing a component, can be viewed as a “switch” that is “closed” when the block is operating and “open” when the block is failed. System is operational if a path of “closed switches” is found from the input to the output of the diagram [17]. Blocks can be connected in series (to represent components that are all required for system functioning), in parallel (to represents blocks of which at least one is required), in a $k - of - n$ structure (when at least k out of n components are required). The overall structure can be composed of all of these kinds of connection, leading either to series-parallel RBDs (that can be solved by simple series-parallel reductions) or to non-series-parallel RBDs (that can be solved by state enumeration, factoring, conditioning, or binary decision diagrams (BDDs)).

Fault trees relate combination of basic events to the system failure. It is a graphical representation in which components are connected with each other through logical gates, in a tree-like structure. Failures of a component (i.e., a basic event) or subsystem cause the corresponding input to the gate to become true; when the output of the topmost gate becomes true, the system is considered failed. In their basic version, fault trees use AND gates to connect parallel components, OR gate to connect series systems and $(n - k + 1)$ *of n* gate for *k-out-of-n* components. Extensions to Fault-trees include further gates, such as NOT, EXOR, Priority AND, cold spare gate, functional dependency gate and sequence enforcing gate. Fault trees easiness and their simple graphical representation are the main reasons for their success, which is confirmed by their extensive usage for real and complex systems modelling.

In order to improve their modelling power, several extensions to the fault trees formalism were proposed in the literature, such as Dynamic Fault Trees (DFT) [22], Parametric Fault Trees (PFT) [23], and Repairable Fault Trees (RFT) [24].

STATE-SPACE MODELS

When the accuracy of combinatorial models is not enough to capture the characteristics of the system to be modelled, state space based models can be considered.

Models in this category have a greater modelling power and flexibility than combinatorial models, but the state space analysis may be computationally expensive. This depends on the number of states in the model, since the state space size grows exponentially with the number of components in the system. The space state explosion problem has triggered many studies and significant results have been achieved based on two general approaches: "largeness avoidance" and "largeness tolerance". Largeness avoidance techniques try to circumvent the generation of large models. They are complemented by largeness tolerance techniques which provide practical modelling support to facilitate the generation and solution of large state-space models.

The basic formalism for state-space modelling are **Markovian models**. A Markov process is a stochastic process whose dynamic behaviour is such that probability distributions for its future development depend only on the present state and not on how the process arrived in that state [17]. When the state-space is discrete (i.e., the set of all possible values that can be assumed by random variables of the process is discrete), the Markov process is known as a Markov chain. Markov chains are a fundamental block for state-space analysis.

They have been used to model a wide number of systems, ranging from network systems, protocols hardware components, software/hardware systems and software applications, complex clustered systems, and for analysing any kind of dependability and performance-related attributes (reliability , availability, performability, survivability).

Markovian models can be classified in: **discrete time Markov chains** (DTMCs), when the model adopts a discrete index T (usually representing the time), **continuous time Markov chains** (CTMCs), when T is a continuous index, and **Markov Rewards Models** (MRMs), when the Markov chain also includes a reward rate (or weight) attached to the states in order to derive additional measures (e.g., expected accumulated reward in a given interval).

The model is usually graphically represented by a state-transition graph, which highlights the system states (the nodes) and transitions among them (the edges) labelled by the one-step transition probability value.

When the Markovian memoryless property does not hold (i.e., it does not accurately describe the system being modelled), non-Markovian models (such as **semi-Markov processes** or **Markov regenerative models**), or **non-homogeneous Markov models** are employed, where other distributions are allowed. The accuracy they add to the model is of course paid in terms of complexity in management, parameterization and solution.

Markovian analysis consists of some basic steps to be carried out: abstraction of the physical system, construction of a Markov model, setting up and solution of ordinary differential equations (for transient solution) or linear equation (for steady-state solution). When the number of states is very large, this becomes a tedious, complex and error-prone procedure. With the increasing size of systems, this problem led in the late 1980s to the introduction of new formalisms and tools. One of this has been particularly successful, due to its ability to concisely represent a complex system in an intuitive fashion: this is **Stochastic Petri Nets**

(SPNs). A fundamental feature of SPNs is that there is a direct mapping between them and CTMCs, which allows designers to model their system by the more intuitive SPN formalism, and then to automatically translate it into a CTMC to be solved (by proper tools, such as SPNP, DSPNExpress, GreatSPN, and SHARPE). Similarly, stochastic reward nets (SRN), that are the extension of SPNs with the addition of rewards, can be mapped onto Markov Rewards Models.

Motivated by their representational power (their graphical representation is also particularly suited to model parallel architectures, concurrent programs, synchronization problems and multiprocessor systems) and their solution capability as Markovian models, researchers defined several variants of stochastic Petri nets, well-suited to particular application needs or solution methods (**Generalized SPNs, Stochastic Activity Networks (SANs), and Coloured Petri Nets (CPN)**).

HYBRID MODELS

Non-state-space models (e.g., RBDs, FTs) are undoubtedly efficient to specify and analyse, but the independent assumption on which they rely on may be too restrictive for many practical situations. On the other hand, Markovian models provide the ability to model systems that violate this assumption, but at the price of a state space explosion. To cope with this problem a number of modelling approaches have been proposed to avoid the generation of a so-wide state space (i.e., the mentioned "largeness avoidance" approach). This is obtained by hybrid modelling. The main idea is to hierarchically compose/decompose the system and construct models accordingly: state-space methods are used for those parts requiring dependences modelling, whereas combinatorial methods are used for the parts that can be assumed independent.

Several research works have been published, that try to combine the advantages of combinatorial and state space based analysis methods: typically, minimal subsystems/components are isolated and treated by state-space methods, and then they are combined in a FT-like structure, exploiting combinatorial analysis techniques at the overall system level (these works also fostered proposals to extend the original FT formalism in order to express dependencies by using an FT-like language).

Example of works adopting hybrid modelling are in [25][26][27][28][29][30][31]. For instance, authors in [27] modelled a SIP Application Server configuration on WebSphere, by using a set of interacting sub-models of all system components capturing their failure and recovery behaviour. A service reliability analysis adopting a hierarchical model is presented in [26]; the hierarchical modelling is mapped to the physical and logical architecture of the grid service system and makes use of Markov models, Queuing theory, and Graph theory to model and evaluate the grid service reliability.

MEASUREMENT-BASED APPROACH

Models are extensively used for dependability attributes evaluation, especially for reliability analysis. However, they may be not accurate enough, when the input parameters values are not representative of the real system behaviour.

Measurements-based approach may allow for more accurate results: it is based on real operational data (from the system or its prototype) and the usage of statistical inference techniques. It is an attractive option for assessing an existing system or prototype and constitutes an effective way to obtain the detailed characterization of the system behaviour in presence of faults. However, since real data are needed, it is not always possible to apply this approach, because data may be not available. Moreover, just relying on measurement-based approach does not yield insight into the complex dependencies among components and does not allow system analysis from a more general point of view. It is often more convenient to make measurements at the individual component/subsystem level rather than on the system as a whole [32], and then to combine them in a system model.

An overview of experimental approaches to dependability evaluation is in [33]. Although the most of papers use either the model based or the measurement based approach, some papers use a combined approach [34], [35]. An online monitoring system combining both the approaches towards system availability evaluation is in [36], [37]. Models parameterized by experimental data are also used in [38] for software aging analysis, in which Markov chains are used to model both system states and workload states, and transition probabilities and sojourn time estimates were obtained by using real experimental data.

1.2.3 Black Box vs. Architecture-based Reliability Analysis

Regardless to the modelling formalism that can be adopted, reliability analysis approaches can be distinguished with respect to their objective. A first class of models, referred to as **"black box"** models, aims to evaluate how reliability improves during testing and varies after delivery; a second class of models focuses mainly on understanding relationships among system components and their influence on system reliability. These are often referred to as **"architecture-based models"**.

Traditionally, approaches to analyse software reliability are "black box", that is, they treat the software application as a monolithic whole by modelling its interactions with the external environment. The black box approach ignores information about the internal structure of the application and neglects relationships among system components. It is based on (i) collecting failure data during testing, and (ii) calibrating a software reliability growth model (SRGM) using such data. This model is then used for prediction in the operational phase (to predict the next failure occurrences based on the trend observed during testing) and/or in the testing phase of successive system releases to determine when to stop testing.

Many SRGMs have been proposed in the literature. Commonly used models adopt a non-homogeneous Poisson process (NHPP) to model the software reliability growth during the testing phase.

NHPP models are distinguished by the form of its mean-value function $m(t) = E[N(t)]$, where $N(t)$ is the number of failures occurring in the time interval (0,t], or equivalently by its integral $\int_0^t m(x)dx$, named failure intensity. One of most successful models was proposed in 1979 by Goel and Okumoto (GO)[39]. It assumed an exponential failure intensity function $\lambda(t) = b(a - m(t)) = abe^{-bt}$, where a is the expected number of faults to be detected if the testing is carried out indefinitely, b is interpreted as the failure occurrence rate per fault (that, in this context, is named hazard rate h(t)). By allowing h(t) to be not constant, other models are obtained. A generalized version of GO model is derived by using the Weibull distribution [40], $h(t) = bct^{c-1}$. Gokhale and Trivedi [41] proposed a log-logistic distribution for h(t) in order to caputure the increasing/decreasing behaviour of the failure occurrence rate per fault.

Many other models have been proposed, also capturing debugging activity or infinte-failure behaviour [42] (i.e., models assuming that an infinite number of faults is detected in infinite testing), and several tools have been developed to deal with fitting and parametrization of the most suitable model for a given set of failure data (such as SREPT, SMERFS, SoRel and CASRE). However, the main problem of these models remain the inability to consider internal system components and their interactions.

To cope with these limitations, architecture-based approaches have been proposed in the literature. This kind of models have gained importance since the advent of object-oriented and component-based systems, when the need to consider the internal structure of the software to properly characterize its reliability has become important. This led to an increasing interest in the architecture-based reliability and performance analysis [43, 44, 45, 46]. Architecture-based models can be categorized as follows [47]:

- **State-based** models use the control flow graph to represent software architecture; they assume that the transfer of control among components has a Markov property, modelling the architecture as a Discrete Time Markov Chain (DTMC) a Continuous Time Markov Chain (CTMC) or semi Markov Process (SMP).

- **Path-based** models compute the system reliability considering the possible execution paths of the program.

- **Additive-models**, where the component reliabilities are modelled by non-homogeneous

Poisson process (NHPP) and the system failure intensity is computed as the sum of the individual components failure intensities.

State-based models can be further categorized into **composite** and **hierarchical** models [48]. In the former, the software architecture and the failure behaviour of the software are combined in the same model, while hierarchical approach separately solves the architectural model and then superimposes the failure behaviour of the components on the solution. Although hierarchical models provide and approximation to the composite model solution, they are more flexible and computationally tractable. In the composite model, evaluating different architectural alternatives or the effect of changing an individual components behaviour is computationally expensive. Unlike hierarchical models, they are also subject to the problem of stiffness [49]. To cope with the accuracy gap between hierarchical and composite models, Gokhale and Trivedi [50] included the second-order architectural effects in hierarchical models.

In this thesis we use both black box and architectural (state-based) models for addressing the problem of conveniently allocating resources available for verification to various parts of the system. The solution proposed adopts a combination of both kind of models, in that it tries to leverage state-based models ability of more accurate capture of the architecture, by combining it with the NHPP models ability in relating reliability and testing time.

1.3 Software Verification

1.3.1 Basic Concepts

As any other engineering discipline, building high quality software products requires construction activities to be complemented by intermediate checking activities and verification of the final product. In particular, *verification is responsible for assuring that the software is rightly being developed, i.e., it is the set of activities to provide adequate confidence that the software product conforms to its specified requirements.*

Compared to other engineering fields, software verification cannot rely on well-established and trained processes able to outline what steps should be carried out in any case and what is the final result. This is mainly due to two reasons. First, software engineering is a relatively young discipline as compared to other engineering fields, and it is simply not mature enough to cope with issues raised by software products' needs. Second, software seems to be inherently more complex than other engineering products, since:

- Software engineering rapidly evolves, continuously proposing new development techniques and approaches, which also bring new challenges to be addressed for verification (i.e., object-oriented development, or multithreading);
- Structure of a software system changes quickly over time (and also deteriorates),
- Faults affecting software, as shown in the previous section, may be difficult to identify, their distribution cannot easily be foreseen, and their activation can be even non-deterministic;
- Software is intrinsically non-linear, as exemplified by [1]: *"if an elevator safely carry*

a load of 1000 kg, it can also safely carry any smaller load, but if a procedure correctly sorts a set of 256 elements, it may fail on a set of 253 or 53 or 12 elements, as well as on 257 or 1023";

- Finally, software systems have many different, often incompatible, quality requirements, that make verification activities impossible to be determined once for all.

This makes cost devoted to software verification to be notably high. Hence, engineers have to effectively weave quality assurance and improvement activities in the development process in order to measurably guarantee that the software meets the requirements specification. Verification starts at very beginning of the software development, from requirements elicitation (when qualities to be assured are identified), to coding, delivery and maintenance. It is not the set of activities to apply once the system is developed.

The activities usually carried out include inspection, static/dynamic analysis, and testing. In the following, a brief overview of issues related to the process activities and to techniques usually involved in verification plan is presented.

1.3.2 Verification Process

The final quality of software depends on many intertwined activities, involving specification/design as well as verification activities, and on how these activities are mapped into the specific organization. Verification process (i.e., the set of activities and responsibilities to ensure the desired quality) is therefore tightly related to the development process and to organizational factors. It provides a framework for selecting and arranging activities to pursue quality objectives within cost constraints.

Activities and actions to be taken and their schedule in a verification plan depend on:

- *the application domain that often determines the quality requirements to be achieved (e.g., standards for safety critical system imposes specific reliability goals);*
- *the development environment (e.g., the adopted development process affects the activities and their schedule);*
- *the structure and size of the organisation, which determines the responsibilities and roles assignment policy [1];*
- *the size of the system to be developed, that determines what activities are most suited;*
- *the cost constraints to be met, which may affect design choices, such as the inclusion of Off-The-Shelf (OTS) components, and in turn verification;*
- *once high-level design is complete, the system architecture also affects the resources allocation policies, and hence verification effectiveness.*

The ultimate goal of a verification plan is to determine i) what activities will be carried out and in what order; ii)how they will be mapped and related to the development activities; iii) how efforts will be allocated; and iv) how the achieved quality can be assessed.

Improving a process means improving the ability to systematically answer such questions, so that plans can be effectively be produced within an organisation across several products.

Companies are required to learn from past for deriving useful guidance in planning verification. Hence the retrospective analysis of a software product development is a key step to improve the process. The most important techniques to this aim are the Orthogonal Defect Classification (ODC), the Root Cause Analysis (RCA), growth modelling, and fault proneness models for efforts allocation.

ODC attempts to compare the patterns of detected defect types to activities across the development cycle, in order to reveal process flaws. It has been described in section 1.1.2;

RCA aims to identify the root cause of the most important classes of faults (in terms of severity and types) in order to eliminate process faults. It identifies process deficiencies by applying four fundamental steps: what (i.e., what are the faults that occurred), when (i.e., when they occurred and when they were found), why (i.e., what is the root cause of the fault) and how (i.e., how they can be prevented by intervening in the process).
Differently from ODC, it does not have a predefined set of categories, since it does not aim to compare different classes of faults over time. The idea is that RCA eliminates progressively the causes of the most important faults at a given time; but as they are eliminated, they lose importance. Hence a static classification would not make sense. Moreover, the most important fault types depend on the specific current project/process, therefore it may not be always the same.

Growth modelling has been briefly discussed in section 1.2.3, when models for reliability were discussed. They make use of past data (or sometimes of in-process data) to calibrate models of reliability growth over time, which enable the evaluation of testing effectiveness in achieving the desired reliability in a given time. However, this technique does not capture the semantics behind the defects and therefore cannot suggest significant improvement actions.

Fault proneness models allow to exploit historical data in order to identify, through empirical analyses, software modules more prone to contain faults. This allows to better allocate efforts to the system modules. Such models have recently re-gained attention, due to the wide availability of data in large bug tracking databases and version control systems.

This thesis proposes means for improving verification process effectiveness, specifically when the quality goal to be achieved is reliability. Solution that we propose for cost-effective verification uses some concepts of these techniques: ODC is used as fault classification scheme to evaluate the influence of defect types on the selection of verification techniques; it is used in conjunction with fault proneness models, that we adopted to evaluate the tendency of software modules to be affected by faults of different types. This is described in section 4 and 5. Reliability growth models are instead used in the allocation of verification resources to various system's components/subsystems, as showed in section 3.

Note that all these techniques are not mutually exclusive. More techniques can suitably be adopted, standing the availability of economical resources. Similarly, integrations of what we propose in the next chapters with one or more of these techniques can be also envisioned.

1.3.3 Verification Techniques

Planning verification requires the selection of a set of techniques to employ. Techniques for verification include testing and analysis. Pezze' and Young [1] classify techniques according to their inaccuracy with respect to the ideal optimal verification (i.e., the exhaustive testing or perfect verification by logical proof). They distinguish techniques that are pessimistically inaccurate, meaning that they do not guarantee to accept a program even if the program does possess the property being analysed (e.g., the most of analysis techniques) or optimistically inaccurate, i.e., they may accept some programs that do not posses the property (testing techniques are optimistic). A third dimension of inaccuracy are "simplified properties", that are the techniques (mainly static analysis techniques) substituting a property that is more easily checked or constraining the class of programs that can be checked.

There is an extensive body of literature about testing and analysis techniques, that is not possible to address in this thesis. Techniques are usually conceived to pursue one specific goal, to improve a specific quality attribute or to verify a particular property. They may be suited for a particular phase of development cycle, for some specific kind of systems and/or for different system sizes.

In the following a brief description of some testing and analysis techniques more relevant for mission-critical systems (the target of this thesis) is given.

Functional testing, along with structural testing, is the most adopted testing technique.

It derives test cases from program specifications and does not require any knowledge of system's internal code (it is therefore also known as black-box testing). Test cases are derived systematically, i.e., not randomly. Usually, the input space is partitioned into a set of classes. Test cases are picked up from these classes and from their boundaries (ideally, classes should include inputs that fail often or do not fail at all). Although functional testing can be used in unit testing, they are more often used for system tests. Combinatorial approaches to generate test cases may also allow to systematically combine input values, leading to what is called ***combinatorial testing***.

Test cases can be picked up from input space also randomly, as if faults were distributed uniformly across the input space (***random testing***). Although this criterion may seem inadequate, several studies have showed that random testing have performances comparable with functional testing, since the generation of test cases is much faster than any other technique.

Structural testing complements functional testing. It derives test cases based on the structure of the code, with the goal of covering the greatest "part" of the program code. The part of the program to be covered is determined by the adequacy criterion, which differentiates the various versions of structural testing. It may be expressed by "executed statements", branches of the control flow graph (CFG), conditions, paths, or a combination of them (one of the most successful one is the modified condition/decision criterion, MC/DC). Since it requires the knowledge of the code, it is often referred to as "white box" testing. For this reason it is used immediately after implementation, i.e., for unit testing.

Another widely adopted criterion is "data flow". Data flow testing generates test cases in order to cover the flow of data between data definition and its usage. Some criteria are the coverage of all possible definition-use (DU) paths (i.e., a path on the CFG starting from a definition to a use of a same variable, in which value is not replaced on the path) or DU pairs (i.e., a pair of definition and use for some variable, such that at least one DU path exists from the definition to the use).

Other than these techniques and their numerous variants, more specific and narrowed techniques have been proposed. The spreading of model-based development favoured a large-scale adoption of ***model-based testing***. Model-based testing is a technique where the runtime behaviour of an implementation under test is checked against predictions made by a formal specification, or model. Models can be developed by using several formalisms (grammars, sets, states), but they are typically given in a finite state machine (FSA)-like form. Test cases may be generated to cover "All transitions", "All states", "Random walks" , "Shortest paths first", "Most likely paths first", etc.. ***Fault-based testing*** (in which faults are injected in the program) may have different purposes: it may be used to evaluate the quality of test suite in detecting faults (as in mutation testing), to evaluate the quality of fault tolerance mechanisms (e.g., fault-injection based testing) or simply to analyse the system behaviours in the presence of faults. Testing based on fault injection is mainly used in mission-critical systems.

Robustness testing generates test cases aiming to evaluate the system behaviour under exceptional conditions. Thus, it deliberately forces the system with unexpected inputs and observes its ability to manage such values. Robustness testing is often used in conjunction with functional testing techniques especially in critical systems, since its purpose is the opposite of functional testing (it has to verify that the system does not do what is not required)

Stress testing evaluates the application's ability to react to unexpected loads; even if the similarity with robustness testing is evident, stress testing does not use "exceptional inputs"; it uses normal input values, but with excessive load. It lies in between robustness and functional testing.

Statistical testing explicitly tests software for reliability rather than for fault detection. It is based on the definition of an operational profile that should accurately describe the system usage at the operational time. Test cases are generated from this operational profile and software is tested and amended until that level of specified reliability is reached. The main problem with statistical testing is that the operational profile may not be an accurate reflection of the real use of the system and that a statistically significant number of failures should be caused to compute the reliability (but highly reliable systems will rarely fail). Hence, this technique will probably increase rapidly the reliability at the beginning of the testing, but will hardly boost reliability beyond a certain limit, if used alone.

Automated analysis techniques are conceived to exhaustively check properties that are difficult to test (e.g., faults that are rarely activated). They can in some cases replace (or support) the manual ***inspection***, which however remain a key technique for some classes of faults. Analysis can be either ***static or dynamic***. Static analysis examines program source code to verify the property of interest and spans over the entire execution space. Hence it generates many false warnings and may require too much time, but it may potentially detect any kind of faults. For this reason it is often used to verify some specific property and on some critical parts of the system. A relevant example is software ***model checking***, which recently re-emerged thanks to new techniques able to cope with the known problem of state space explosion. It is able to verify several kinds of properties, expressed in temporal logics, on a model of the system (that can be derived from specification or extracted from source code). They are particularly suited to detect execution paths violating critical properties, such as safety or liveness, that are difficult to detect by testing. Other static and dynamic analysis techniques are adopted for more specific purposes. For instance, ***point-to analysis*** is a code analysis technique that tries to understand which pointers (or references) can point (refer) to which locations. It is often part of a more complex and ***shape analysis***, that aims to verify properties on dynamically allocated structures. ***Value flow analysis*** refers to a family of techniques (coming from compiler theory) that try to detect computations producing always the same value: these are grouped in new namespaces on which a classical data-flow analysis can be carried out. They can be either path-sensitive or path-insensitive. Another well-known analysis technique is ***slice analysis***, introduced by Mark Weiser. A slice is a part of the program that may affect the values computed at some point of interest, referred to as a slicing criterion. Slicing refers to the computation of such slices, exploiting data flow and control flow dependences. It is useful for debugging purposes, since it eases the location of fault sources. It can be either static, If it uses only static information, or dynamic, if also uses runtime information. Other static analysis techniques that worth to be mentioned are the ***software change impact analysis***, to assess the effects of changes in the program, cluster analysis used in the observation-based testing, and experimental analysis, which aims to characterize the effect of one or more independent variables on one aspect of the program (also used in Delta Debugging implementations).

Dynamic analysis techniques examine program execution traces to verify properties: hence it does not examine the execution space exhaustively, but the most representative executions. It is of course less expensive than static analysis. ***Memory analysis*** instruments the program to trace memory usage and detect improper memory access, incompatible with the current state (e.g., read from uninitialized memory locations). An example of tools for dynamic memory analysis is Purify from IBM. ***Lockset analysis*** attempts to detect violation of the locking discipline to prevent data races. ***Behavioural analysis*** techniques aim to extract the system behavioural model from execution traces that can then be used for comparison with real execution. For instance, they can be used to detect differences between the actual behaviour and the modelled one, inferring the cause and potentially uncovering faults. Another usage example is for regression testing (i.e., compare different versions), or component-based testing (i.e., compare the behaviour of component in different contexts). Other verification techniques that it is worth to mention are ***theorem proof*** and ***symbolic execution***, which are used in rare case for very small piece of code or in conjunction with other techniques. For instance, symbolic execution may be used for ***symbolic testing***. The key principle of symbolic execution is to summarize the values of variable with few symbolic value and then verify properties on this “reduced” program version. Although the reduced set of possible states, symbolic execution still turns out to be inapplicable for medium/large programs. Symbolic testing uses symbolic execution principle, but it does not run all the possible state space; it cuts off paths up to a given depth according to heuristic criteria and verifies few critical properties. The *Prefix* tool is an example of symbolic testing application, which aims to solve memory-related problems, such as invalid pointers, faulty allocations, uninitialized memory and improper operations on resources (e.g., on files).

Chapter 2

Improving Verification Effectiveness for Reliability

This chapter surveys the existing approaches that engineers can rely on to plan an effective reliability-oriented verification. When coping with high reliability demands and tight time/cost constraints, one fundamental aspect concerns the identification of the most critical parts of the system, i.e., the major contributors to its unreliability. This is crucial to conveniently distribute efforts for verification. Moreover, even suitably allocating efforts, an important choice regards the verification techniques to adopt in the plan. Engineers should know what techniques most impact the final reliability and how the system being tested influences their performances. Studies comparing techniques could be important to drive this selection. This chapter discusses past work and open challenges regarding these issues. Relevant studies which solutions, methods and results are either useful for the definition of the proposed approach or for comparison purpose only are reported. The problem of software aging, which is a phenomenon that this thesis intends to consider in the testing phase instead of operational time, is also discussed. The chapter will then describe the contribution provided by this thesis to go beyond the current state-of-the-art.

2.1 Efforts Allocation Models

Approaches to allocate efforts to system's software modules/components/subsystems aim to distribute resources available for verification in a cost effective manner. The principal way to cope with this problem is to use a model: typically the objective is to obtain some kind of predictions (e.g., faults content) from system features that are then used to assign resources accordingly. Note that resources identifying the testing effort can be intended as *person/months, number of test cases, CPU testing time, calendar testing time*, and other metrics. As described in the next chapter, *Testing Effort Functions* (TEFs) can be used to describe the testing effort-testing time relationship, and be suitably integrated in the models. In the following subsections solutions proposed in the literature are surveyed.

2.1.1 Modelling objective: Fault removal vs. Reliability

The task of identifying the parts of the systems that should receive more or less attention in the verification phase mainly depends on what specific purpose the verification has to serve. Typically, verification aims to detect as many faults as possible in the software. Hence criteria to distribute efforts should reflect this need: allocating a greater effort on software modules/components/subsystems expected to contain more faults. Towards this aim, much research has been produced in the past on fault-proneness models, i.e., models able to estimate the content of faults of software modules based on several software metrics.

Less commonly, verification goal is to improve the system reliability. Note that detecting more faults does not imply improving the reliability. Reliability improvement requires the removal of those faults which occurrence at operational time is more frequent. A testing session can therefore remove fewer faults than another but deliver system with higher reliability.

In this case, criteria to allocate effort do not look for modules with higher content of faults.

Rather, modules most affecting system reliability have to be sought, i.e., the ones most critical from reliability perspective. This is not trivial. Research area closest to this need is the one dealing with the well-known problem of “Reliability Allocation”.

FAULT PRONENESS MODELS

As far as fault-proneness models is concerned, several studies related in the past software metrics to observed faults in a given number of samples (i.e., software programs) [51]. Statistical techniques are usually adopted in these papers in order to build regression models that allow one to predict faults content starting from metric values.

In [52] and [53] Object-oriented metrics were proposed as predictors of faults density. A more recent study [54] empirically validated three OO metrics suited for their ability to predict software quality in terms of fault-proneness: the Chidamber and Kemerer (CK) metrics, Abreu’s metrics for object-oriented design (MOOD), and Bansiya and Davis’ quality metrics for object-oriented design (QMOOD).

The study in [54] presents a survey on eight empirical studies showing that OO metrics are significantly correlated with faults.

Other studies using metrics are in [56, 57], which investigated design metrics able to predict modules more prone to failures. In [58], authors used a set of 11 metrics and an approach based on regression trees to predict most faulty modules. In [59] authors mine metrics to predict the amount of post-release faults in five large MicrosoftÕs software projects. They adopted the statistical techniques known as Principal Components Analysis (PCA) in order to transform the original set of metrics into a set of uncorrelated variables, with the goal of avoiding the problem of "multicollinearity" (i.e., an inflated variance evaluation caused by the strong correlation among predictors). Authors also suggest a procedure to build powerful regression models.

The work in [60] uses logistic regression to relate software measures and fault-proneness for classes of homogeneous software products. Authors in [61] extend this study, reporting an empirical analysis of the validity of multivariate models for predicting software fault-proneness across different applications. Some statistical techniques have been investigated by Khoshgoftaar, who applied discriminant analysis with Munson [62] and logistic regression with Allen, Halstead, Trio, and Flass [63].

In many cases, common metrics provide good prediction results also across several different products. However, it is still quite difficult to claim that a given regression model or a set of regression models is general enough to be used even with very different products, as also discussed in [59]. On the other hand, they are undoubtedly useful within an organization developing one specific class of systems, and that in its process iteratively collects faults data.

RELIABILITY ALLOCATION

Lot of work in the past considered the optimal allocation of reliabilities to minimize a cost function related to the design or to the verification phase. Although much initial research dealt with hardware systems (e.g. the series-parallel redundancy-allocation problem has been widely studied [66, 67, 68]), also software systems received attention.

Most of work in the software area coped with the design phase, in which the goal is to select the right set of components with a known reliability and the amount of redundancy for each one of them, minimizing the total cost under reliability constraint [69], or maximizing the total reliability under cost constraint [70, 71, 72] (more specifically, this is a redundancy reliability allocation problem). In some cases they also considered the redundancy strategies and the hardware. For instance, the work in [70] defines a model comparing different redundancy strategies (the *N-version programming* on a single node, *the N-version programming* on more nodes and *Recovery-Block* strategy) at different level, giving as output the best redundancy strategy to achieve the required reliability. In this case, the costs defined in the objective function are considered already known and constants.

When redundancy is not considered, the reliability allocation problem can still refer either to the design or to the verification phase. For instance, authors in [73] proposed an economic model to allocate reliabilities during the design phase, minimizing a cost function, which depends on fixed development costs and on a previously experienced failure decrease cost.

The work in [74] also refers to the design phase and authors define a general-behaviour cost function to relate the costs to the reliability of a component. Cost in the design phase can refer to development (i.e., how much development resources should be employed to produce a component with reliability R), but more commonly it refers to the acquisition (i.e., how much buying a component with certified reliability R costs). Hence the problem in this case is to decide the best policy to obtain the desired overall system reliability by acquiring (or developing) components at minimum cost.

The problem in the verification phase is quite different. Here components reliability has to be achieved by testing activity, and predicting this is indeed more challenging. The question is: *how much testing should each component receive in order to meet the overall reliability goal at minimum testing cost?*

Contrarily to the design, for this problem not many papers appeared in the literature. Among these, authors in [75], proposed an optimization model with the cost function based on well-known reliability growth models. Using growth model in this case is essential, since they are able to describe relationship between testing time (more generally testing resources, such as man/month, calendar time, number of test cases) and reliability growth that testing produces.

Authors in [75] and [76] also include the use of a coverage factor for each component, to take into account the possibility that a failure in a component could be tolerated (but fault tolerance mechanisms are not explicitly taken into account, and the coverage factor is assumed to be known). The authors in [64] also try to allocate optimal testing times to the components in a software system (here the reliability-growth model is limited to the Hyper-Geometric (S-shaped) Model).

Some of the cited papers [70, 75, 76] also consider the solution for multiple applications; i.e., they aim to satisfy reliability requirements for a set of applications. To solve such models typically heuristic approaches are needed. Authors in [77] present a recent overview on solution techniques for the reliability allocation problem.

2.1.2 Explicit vs. Implicit Architecture

Effort allocation approaches necessarily need to consider in some way the system architecture, i.e., they need to characterize the system in terms of components and their interaction. This means that the way in which components interact at runtime should also be considered: a component with high fault content but rarely used could be decided to receive less testing than a less faulty but used component. However, almost all the described solutions consider the system architecture only statically, as sum of components. Fault-proneness models are not conceived to take into account also this aspect: they assume that components with more faults must receive more testing, regardless their runtime usage.

Among reliability allocation models, none of the cited papers explicitly considers the architecture of the application. Most of them belong to the class of the so-called additive models [47] (presented in section 1.2.3), in which the system failure intensity (and reliability) is computed as the sum of the individual components failure intensities. In other words, components usage is not contemplated in the model. Work in [73] and [78], as well as [79] and [70], by partially taking into account the utilization of each component with a factor supposed to be known (i.e., engineers should assign a value to a factor indicating the component utilization; that is not easily derivable). Moreover, among these, only Everett [79] refers to the verification phase. There are, however, other ways to describe a software application, which can explicitly consider the architecture and components usage, and lend themselves to an easy integration with the other aspects of interest for reliability modelling, such as the inclusion of fault tolerance mechanisms. They have been briefly introduced in section 1.2.3 (the state-based models and the path-based models) and have been adopted in this thesis to overcome the limitations of existing models. Section 2.4, and then chapter 3, describes solution we adopt for efforts allocation.

2.2 Comparing Verification Techniques

To select the best technique for a product, a deep knowledge about verification techniques and their relationship with the objective of verification is required, i.e., how they can contribute to achieve the pursued goal.

A reasonable way to gain this knowledge is to compare techniques, or similarly to exploit results of past comparisons. In the following subsections, studies about comparison among verification techniques are presented.

2.2.1 Comparison criterion: Fault Removal vs. Reliability

Several studies in the literature dealt with the comparison of software verification techniques, since eighties. Very often, studies referred to classical testing and analysis techniques, like functional testing, various forms of structural testing, and manual inspection. One of the main issues addressed in these works is the appropriateness of the adopted comparison relations: the first question authors attempted to answer is "how to correctly compare the effectiveness of testing strategies?". One of the first comparison relations was used, in 1982, by Rapps and Weyuker [80, 81], known as the "subsumption relation", where a criterion $C_1 subsumes$ a criterion C_2 if for every program P, every test suite that satisfies

C_1 also satisfies C_2 .

They compared several data-flow and control-flow test case selection strategies by this relation. However, this criterion was recognized to be inadequate (many testing techniques were incomparable using subsumption) or even misleading, as pointed out in [82].

Successively, the "power relation" was introduced by Gourlay [83], which defined a criterion C_1 to be at least as powerful as C_2 if for every program P, if C_2 detects a failure in P , then so does C_1. Although this made a positive step, the incomparability problem was not solved.

The introduction of the "BETTER relation" attempted to address the problems of both the previously introduced relations [84]. Authors first defined the notion of a test case being *required* by a criterion C to test a program P , if every test set that satisfies C for that program must include that test case. Then, they defined the relation, stating that criterion C_1 is BETTER than criterion C_2 if for every program P , any failure-causing input *required* by C_2 is also required by C_1. Although this attempt, authors also stated that very few techniques are comparable with this relation, leaving the incomparability problem unsolved.

In [88] authors addressed the problem by a different view, followed then by others [85]. They address the problem by using a probabilistic measure. Along this trend, authors in [86] introduced the *properly covers* and *universally properly covers* relation, which use the probability that a test suite expose at least one fault as metric to compare and rank techniques. This paper represented an important step, since research on this field switched to probabilistic comparison criteria: a test suite or a criterion is not deterministically better than another, but just probabilistically. This better reflects the reality, where the high number of variables makes the detection effectiveness of a criterion not guaranteed to be always better than another: it is just more likely that a criterion detects more faults than another.

It is relevant for this context to point out that none of these studies consider reliability measure to compare techniques. From these papers it is not possible to infer what is the impact that a testing technique has on the final system reliability. As already mentioned, even if a technique is guaranteed to detect more faults than another one, it is not possible to say that reliability improves.

However, although not much research has been devoted to this aspect, few work addressed from a theoretical point of view this issue. In [65], the authors analytically compared what they refer to as debug testing with operational testing with the goal of evaluating which technique produces higher reliability. With debug testing, they intend criteria to select test cases that will cause failures to occur. With operational testing, they intend what we called "statistical testing", i.e., selecting inputs according to a statistical distribution representative of real usage.

2.2.2 Method of analysis: theoretical vs. empirical

Much of the literature, since the eighties, evaluates test cases selection criteria by theoretical comparison. All the cited studies in the previous subsection were about theoretical comparisons.While with the described theoretical relations, authors were able to compare many testing criteria, the main problem of this approach is the practical applicability of results in real development projects. Work in [87] lists a set of assumptions that are difficult to satisfy in practice; the main issue is that the "idealized" versions of compared techniques are never the actual versions used in practice, due to the numerous underlying assumptions. Moreover, the testing environment changes cannot easily be taken into account (e.g., the human variability, due to different expertises, experiences and acquired intuition). On the other hand, there are few empirical studies that compare the effectiveness of testing techniques in real-scale software systems. Many of them used generally small programs, specifically written for experimentation [88, 89, 90, 91, 92]; some examples of large projects are present in [93, 94].

The lack of such studies is mainly due to the difficulties and cost associated with this kind of analysis; however also in [87] it is recognized that there is a profound need for empirical study in this field. The major conclusions are that our current testing technique knowledge is very limited. The work in [96] reports a survey on testing techniques experiments. They considered: random testing, functional testing, control flow testing techniques, data flow testing and regression testing techniques. Conclusions drawn from this analysis are even worse than Weyuker in [87]. They conclude that "there is at present no formally tested knowledge and over half of the existing knowledge is based on impressions and perceptions and, therefore, devoid of any formal foundation".

From reliability perspective, the lack of studies highlighted in the previous subsection is confirmed for empirical studies. The paper in [95] is one of the rare examples in which techniques are compared empirically with respect to the delivered reliability. Experiments were conducted on a moderate-sized C-program (about 10,000 LOC) produced by professional programmers and containing naturally occurring faults. Compared techniques were branch testing, the all-uses data flow testing criterion, and operational testing.

2.3 Software Aging

Software Aging can be defined as a continued and growing degradation of software internal state during its operational life. This problem leads to progressive performance degradation, occasionally causing system crashing. Due to its cumulative property, it occurs more intensively in continuously running processes that are executed over a long period of time. It is usually due to the already mentioned *"aging-related"* bugs.

Recent studies showed that a large number of software systems, employed also in business-critical or safety-critical scenarios, are affected by Software Aging. The Patriot missile defence system employed during the first gulf war, responsible for the scud incident in Dhahran, is perhaps the most representative example of critical system affected by software aging [7], in which crash was caused by an accrued round-off error in the calculation of target's expected position. Users of the system were warned that *very long runtime* could negatively affect system's targeting capabilities. Unfortunately, they were not given a quantitative evaluation of such "very long runtime", thus leading to the famous incident in which 28 people were killed.

Typically, this "long-running" phenomenon is not addressed in the testing phase, as we intend to do in this thesis. It is instead dealt with approaches that aim to predict the time to failure (in this case, it is the time to exhaustion), and then at operational time, to activate proper actions (that are usually referred to as “rejuvenation techniques”) with an optimal schedule (i.e., neither too early because it is useless and expensive, nor too late, but just before the failure occurrence). In the following such approaches are briefly described. They are categorized in analytic modelling and measurements-based approaches.

ANALYTIC MODELLING

Analytic modelling generally determines the optimal rejuvenation schedule starting from models. Typically, to model the software system affected by aging, stochastic processes representing system’s states are adopted. In [97] a Markov Decision Process (MDP) is used to build a software rejuvenation model in telecommunication system. In [98] Markov semi-ReGenerative Processes (MGRP), in conjunction with Stochastic Petri Nets (SPN), are used to build a simple but general model for estimating the optimal rejuvenation schedule in a software system. Stochastic Deterministic Petri Nets (SDPN) are employed in [99], to build a model to analyze the performability of cluster systems under varying workload. Non-homogeneous, continuous time Markov Chains are instead used in [100]. Semi-Markovian Processes have also been used to model proactive fault management in [101].

Different probabilistic distributions were chosen for time-to-failure (TTF). Some papers, such as [98] are restricted to an hypo-exponential distribution, whereas other papers, such as [102] employs more general distributions for TTF, like the Weibull distribution. However these TTF models are not able to capture the effect of load on aging.

A common shortcoming with analytic modelling is that the accuracy of rejuvenation schedule deeply depends on the goodness of the model (i.e., how good the stochastic model used to represent the system approximates the real behaviour of the system) and on the accuracy of the parameters used to solve the model (e.g., failure rate distribution expected value, probability of transition from the “steady” state to the “degraded” state). Trivedi and Vaidyanathan in [38] addressed this problem, by building a measurement-based semi-markovian model for system workload, defining a set of workload states through cluster analysis, estimating the time to exhaustion (TTE) for each considered resource and state using reward functions, and finally building a semi-Markovian availability model, based on field data rather than on assumptions about system behaviour.

MEASUREMENT-BASED APPROACH

The basic idea of Measurement-based approaches is to directly monitor attributes subject to software aging, trying to assess the current “health” of the system and obtain predictions about possible impending failures due to resource exhaustion or performance degradation. In [105], a measurement-based analysis performed on a set of Unix workstation is reported. A set of 9 Unix Workstations has been monitored for 53 days using an SNMP-based monitoring tool. During the observation period, 33% of reported outages were due to resource exhaustion, highlighting how much software aging is a non-negligible source of failures in software systems. An interesting workload based software aging analysis can be found in [106]. This paper presents the results of an analysis conducted on the same set of Unix workstation of the previous paper, which takes into account also some workload parameters, such as the number of CPU context switches and the number of system call invocations. In this case different workload state are first identified through statistical cluster analysis; then a state-space model is built determining sojourn time distributions; a reward function, based on the resource exhaustion rate for each workload state, is then defined for the model. By solving the model, authors obtained resource depletion trends and TTE for each considered resource in each workload state. Although a consistent number of measurement-based analysis deal with resource exhaustion, only a few of them deal with performance degradation. In [107], Gross et al. applied pattern recognition methods to detect aging phenomena in shared memory pool latch contention in large OLTP servers. In [108, 109], Trivedi et al., analyzed performance degradation in the Apache Web Server by sampling web serverÕs response time to predefined HTTP requests at fixed intervals.

Software Aging in a SOAP-based server was analyzed in [110]. A SOAP-based web server running on top of a Java Virtual Machine, has been stressed with different workload distribution. For each considered distribution, throughput loss and memory depletion highlighted the presence of software aging.

Finally, in [111], Malek et al., propose a best practice guide for building empirical models to forecast resource exhaustion. This best practice guide addresses the selection of both resource and workload variables, the construction of an empirical system model, and the sensitivity analysis.

2.4 Thesis Contribution

When quality manager has to verify a critical system, he has on one hand standard reliability requirements to satisfy, and, on the other hand, a deadline for delivering the system and cost constraints that limit his choices. What he needs is a clear quantitative evidence of consequences of his choices. Especially for large systems, he cannot rely only on his intuition and experience to take crucial decisions. In particular, he should know, for the specific system being developed, what happens if a component is devoted a given testing effort or is tested with a particular set of techniques: what is the impact that these choices have on the final reliability and on the overall cost/time. On the base of such knowledge, possibly refined across several projects, he should then tune the process and setting up a plan for his system, simply following a quantitative reasoning that highlights cost/benefits of each choice. ***Based on these considerations, this thesis proposes a solution to carrying out an effective verification specifically oriented to improve reliability.*** It intends to provide engineers with quantitative means that should be adopted and embedded in their process, to allow them ***conveniently allocating*** efforts and ***selecting techniques*** for the system under testing. These are presented in chapter 3, 4 and 5. A procedure to best use such means is outlined, encompassing the steps that engineers should take to plan verification and tune a process that refines itself across several projects. This is described in chapter 6.

With respect to past research, as discussed in the previous sections, not much work in the past dealt with what we referred to as "reliability-oriented" verification. Solutions to single issues have been proposed, as for instance reliability allocation models or rejuvenation techniques (at runtime) for software aging. However no comprehensive solution looking at the verification process (and choices it implies) has been proposed. Generic solutions for process improvement, such as ODC or RCA described in section 1.3.2, aim to identify process bottlenecks (e.g., ineffective phases), but they are not conceived to address the outlined problems. ***The support to a reliability-oriented verification process in its most critical choices is therefore the major contribution of this thesis.*** In addition, previous sections showed that existing related works addressing single issues also suffer from several limitations. In the following subsection, the thesis contributions with respect to such related work are highlighted.

2.4.1 Comparison

As for ***efforts allocation*** driven by reliability objective, existing models (i.e., reliability allocation models) do not consider important issues for their actual applicability. They often make so many assumptions that their adoption is difficult in practice. For instance none of the solutions described in the section 2.1.1 considers that:

- A model for reliability analysis of a software architecture should explicitly describe the relationships among its components, in order to consider the effects of individual component reliabilities on the system reliability. In other words, the architecture cannot be viewed as the sum of its components; interaction plays a crucial role.
- The reliability of a complex system does not only depend on the application components, but also on the operational environment; thus, a model should also take into account the reliability of the underlying software layers, such as the Operating System. Indeed, due to the intensive and continuous usage of OS services by the application

components, the OS reliability level has a significant impact on the overall system reliability so that it should not be neglected;

- Since in a critical application fault tolerance mechanisms are increasingly adopted, a model should consider the presence of such means of failures mitigation;
- For a model to be useful in practice, it needs to be flexible enough to give detailed answers when the user has a lot of information, and to continue giving useful indications, even though less accurately, when not much information is available.

The solution we propose to quantitatively identify the most critical components of a software architecture includes all of the above-mentioned aspects affecting the reliability of a complex software system. Moreover, it addresses the issue, often neglected, of how retrieving the necessary information to parameterize the model and provide accurate results.

Note that in section 2.1.1, we presented fault-proneness models as potential mean to allocate testing efforts. For the purpose of this work, allocating efforts based on faults content is not the best solution; we need to allocate efforts based on reliability (for this reason we adopted a reliability-allocation solution). However, in this thesis fault-proneness models are very important. Their predictive power has been used in two different (both innovative) ways, detailed in the following, i.e., for characterizing software with respect to fault types (according to the ODC scheme), useful for verification techniques selection, and for the aging characterization of the system.

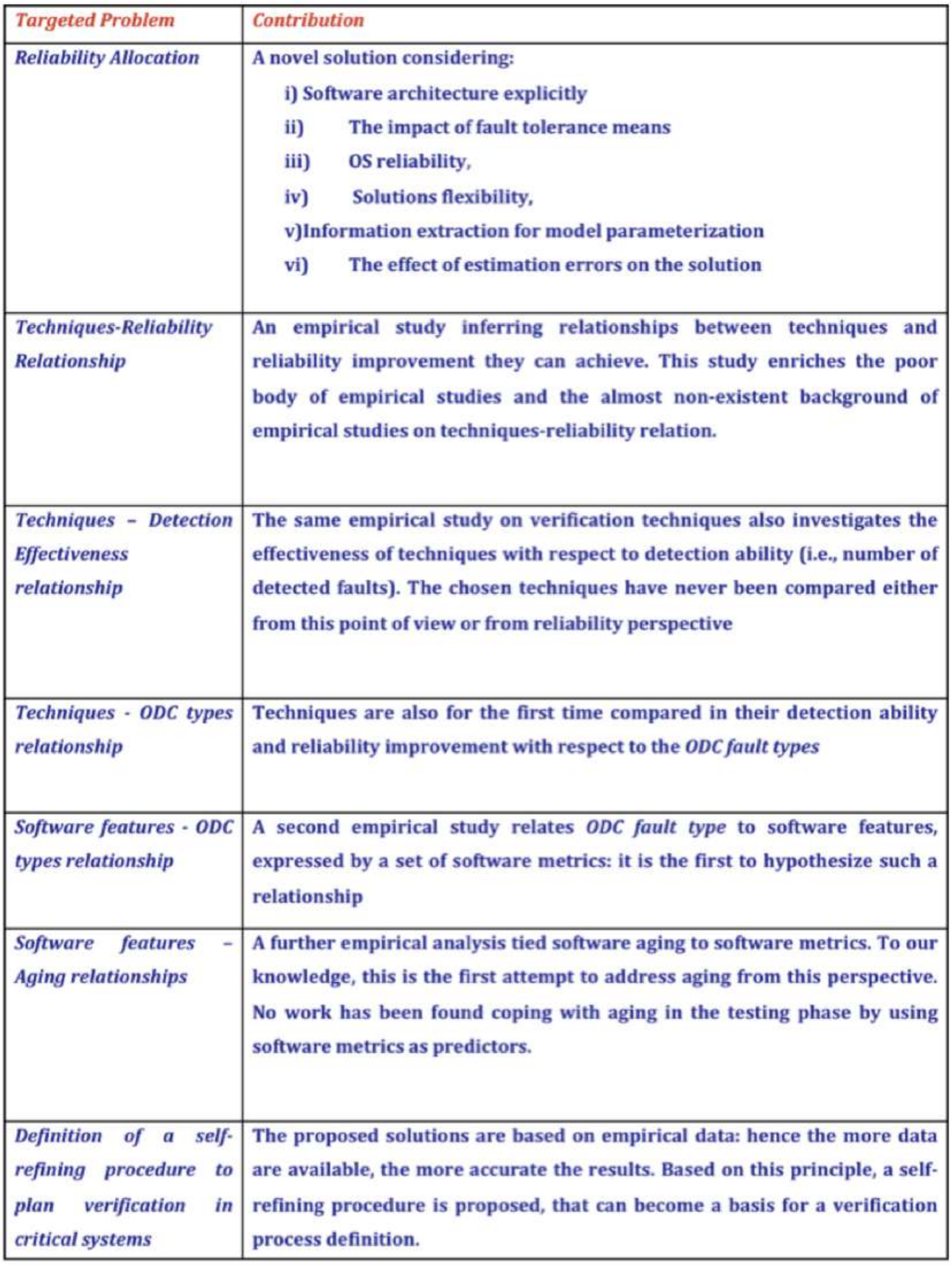
As for ***verification techniques selection***, past studies show that there is very little comparison among techniques from reliability perspective; hence it is not possible to say if a technique is better than another to improve reliability. Moreover, such a statement could be true for a system and not for another. In this thesis we propose a solution to drive techniques selection tailored for the system under test, and according to reliability improvement that they are able to provide in that specific case.

Two empirical analyses have been conceived and carried out towards this aim. First, we attempted to relate some software features to distinct ODC fault types potentially contained in a software system by fault-proneness models; the hypothesis that we investigated is that values of some metrics can be related to the type of faults present in the software. Such a relationship would allow engineers to obtain a preliminarily estimate not only of faults content, but also of faults type possibly present in a module.

Second, four widely used verification techniques have been empirically compared from reliability's point of view and with respect to the distinct fault types (again according to the ODC scheme). Hence, reliability improvement brought by one technique is tied to the kinds of fault present in the software (and to other factors too), which in turn are related to software metrics.

Results of such analyses are tied together: starting from the features of the software system being developed, described by its metrics, engineers characterize the system with the preliminary estimate of faults type and content. Then, they can select techniques by estimating the reliability improvements that a technique may bring in that specific situation (i.e., with those fault types and contents describing the system). The detailed procedure is outlined in chapter 6. In this procedure the problem of ***software aging*** is also taken into account. In particular, we sought relationships, again through concepts coming from fault-proneness models, between software aging and some common software metrics, along with other specific metrics potentially symptomatic of this phenomenon. In this case, the goal is to provide testers with a preliminary indication of what modules should receive more attention (in this, specifically for aging testing).

Summarizing, contributions transpired from the above description target the identified issues for the effective reliability-oriented verification. Figure 2.1 summarizes such contributions, along with further side-effect contributions, with respect to the problem they coped with and to the challenges they addressed. The following chapters will explode each of them, providing further details.



Targeted Problem	*Contribution*
Reliability Allocation	A novel solution considering: i) Software architecture explicitly ii) The impact of fault tolerance means iii) OS reliability, iv) Solutions flexibility, v)Information extraction for model parameterization vi) The effect of estimation errors on the solution
Techniques-Reliability Relationship	An empirical study inferring relationships between techniques and reliability improvement they can achieve. This study enriches the poor body of empirical studies and the almost non-existent background of empirical studies on techniques-reliability relation.
Techniques – Detection Effectiveness relationship	The same empirical study on verification techniques also investigates the effectiveness of techniques with respect to detection ability (i.e., number of detected faults). The chosen techniques have never been compared either from this point of view or from reliability perspective
Techniques - ODC types relationship	Techniques are also for the first time compared in their detection ability and reliability improvement with respect to the *ODC fault types*
Software features - ODC types relationship	A second empirical study relates *ODC fault type* to software features, expressed by a set of software metrics: it is the first to hypothesize such a relationship
Software features – Aging relationships	A further empirical analysis tied software aging to software metrics. To our knowledge, this is the first attempt to address aging from this perspective. No work has been found coping with aging in the testing phase by using software metrics as predictors.
Definition of a self-refining procedure to plan verification in critical systems	The proposed solutions are based on empirical data: hence the more data are available, the more accurate the results. Based on this principle, a self-refining procedure is proposed, that can become a basis for a verification process definition.

Figure 2.1: Summary of Contributions

Chapter 3

Verification Resources Allocation

Efficacy of the verification phase strongly depends on the correct identification of the most critical components in the software architecture, as the available testing resources are usually allotted based on the components' risk levels. Identifying parts of a complex system that are the major contributors to its unreliability is not always an easy task, and consequently the testing resource allocation is most often based on engineering judgement, and hence suboptimal. Some standards and methodologies for critical systems suggest to assign predefined risk levels to components, based on the risk level of the services in which they are involved. This is clearly a rough-grained assignment that merely provides some guidelines for the verification phase. It does not consider the costs and does not provide sufficient modelling basis to let engineers quantitatively optimize resources usage. Existing reliability allocation models are inadequate for the reasons exposed in the previous chapter (section 2.4.1). This chapter presents an allocation model to quantitatively identify the most critical components to best assign the testing resources to them. The chapter first describes the mathematical formulation, then discusses its practical applicability depending on the actual available information about the system, and finally presents its application to an empirical case study.

3.1 Model Overview

Qualitative approaches to identify critical components of an architecture, based on engineers' judgment, do not provide sufficient quantitative information to suitably allocate testing resources and quantify the reliability of the final system. Moreover, they could even lead to wrong assignment (e.g., a less reliable, but rarely used component could affect the total reliability less than a more reliable and frequently used one). Allocation based on quantitative reasoning is essential to answer questions such as:

- How much risk does a component pose to the system?
- Do components at the same risk level have the same impact on the system reliability?
- What is the impact of a change in the reliability of a component on the system reliability?

Most importantly, non-quantitative approaches cannot answer the following question:

- ***What is the reliability that each component needs to achieve in order to assure a minimum system reliability level, and at what cost?***

We believe that a system analysis aiming at assuring the required reliability while minimizing the testing cost needs to be quantitative.

The approach presented in this chapter aims to quantitatively address this analysis. We present an optimization model for testing resources allocation that includes all the aspects affecting the reliability of a complex software system discussed in section 2.4.1.

In particular, to represent the software architecture, we employ the so-called architecture-based reliability model; a Discrete Time Markov Chain (DTMC) type state-based model is adopted. This allows us to explicitly consider the effects of such architectural features as loops and conditional branching on the overall reliability (section 3.2). Moreover, the architectural model encompasses the operating system to consider its reliability and its influence on the application. The proposed optimization model also considers the most common fault tolerance mechanisms (such as restart a component, retry application as recovery mechanisms as also a failover to a standby) that critical systems typically employ (section 3.3.2).

Furthermore, we try to impart the necessary flexibility to the model by: (i) providing different levels of solutions according to the information the user gives as input, and (ii) carrying out a sensitivity analysis in order to analyze the effect of the variation of some parameters on the solution. How to retrieve the information needed for model parameterization is also an addressed issue (section 3.4). Finally, the impact of performance testing time and the second-order architectural effects are also considered for a greater accuracy of the result.

The optimization model is implemented in a prototype tool, which receives the model parameters (e.g., the DTMC transition probabilities) and the user options as inputs and provides the solution by using an exact optimization technique (the sequential quadratic programming algorithm).

The approach is then applied to an empirical case study to verify the accuracy of the prediction abilities (section 3.5). The program chosen consists of almost 10,000 lines of C code, and was developed for the European Space Agency (ESA). Once we built the DTMC architectural model, we predicted the amount of testing needed for each component, in order to achieve a predefined level of reliability. Then the software was tested according to the predicted times and the actual achieved reliability was compared with the predicted reliability. We also evaluated the effect of the error in the parameter estimations on the prediction accuracy.

The model presented in this chapter is reported in [103]. In the following, Section 3.2 describes the adopted architectural model and Section 3.3 illustrates the optimization model. Section 3.4 discusses the possible sources of information about the parameters of the architecture and its components. Finally, Section 3.5 discusses the experiments, followed by the conclusion.

3.2 Architectural Model

The proposed approach adopts a state-based architectural model (see section 1.2.3) to represent the architecture, and NHPP additive models to describe the reliability growth of components with testing time. It tries to leverage state-based models ability of more accurate capture of the architecture, by combining it with the NHPP models ability in relating the reliability and the testing time, as showed in the next Section. In this section the DTMC architectural model is described. Architecture-based models are conceived to relate the behaviour of the system (expressed by some attribute, e.g., the reliability) to the behaviour of its parts. In the literature, the parts of the application under study are often referred to as "components". Although the notion of component is not well defined and universally accepted (except for applications based on component models such as CCM, EJB or DCOM), in the context of architecture-based analysis it is intended as a logically independent unit performing a well-defined function [44]. The level of decomposition, which defines the granularity of components, is an analysis choice, addressed by a trade off between a large number of small units and a small number of large units.

We describe the software architecture by an absorbing DTMC, to represent terminating applications (as opposed to irreducible DTMCs, which are more suitable to represent continuously running applications.) A DTMC is characterized by its states and transition probabilities among the states. The one-step transition probability matrix $\mathrm{P} = [p_{i,j}]$ is a stochastic matrix so that all the elements in a row of P add up to 1 and each of the $p_{i,j}$

values lies in the range [0, 1]. The one-step transition probability matrix with n states and m absorbing states can be partitioned as:

$$P = \begin{pmatrix} Q & C \\ 0 & I \end{pmatrix} \tag{3.1}$$

where Q is an *(n-m)* by *(n-m)* sub-stochastic matrix (with at least one row sum < 1), I is an m by m identity matrix, 0 is an m by *(n -m)* matrix of zeros and C an *(n-m)* by m matrix. If we denote with P^k the k-step transition probability matrix (where the entry *(i,j)* of the submatrix Q^k is the probability of arriving in the state j from the state i after k steps), it can be shown [17] that the so-called fundamental matrix M is obtained as

$$M = (I - Q)^{-1} = I + Q + Q^2 + \cdots + Q^k = \sum_{k=0}^{\infty} Q^k \tag{3.2}$$

Denoting with $X_{i,j}$, the number of visits from the state i to the state j before absorption, it can be shown that the expected number of visits from i to j, i.e., $v_{i,j} = \mathrm{E}[X_{i,j}]$, is the $m_{i,j}$ entry of the fundamental matrix. Thus, the expected number of visits starting from the initial state to the state j is:

$$v_{1,j} = m_{1,j} \tag{3.3}$$

These values are called expected visit counts; we denote them with $V_j = v_{1,j}$. They are particularly useful to describe the usage of each component in the application control flow. We can also compute the variance of visit counts, using M [114]. Denote with $\sigma^2_{i,j}$ the variance of the number of visits to j starting from i. If we indicate with M_D the diagonal matrix with:

$$M_D = \begin{cases} m_{i,j} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

and define $M_2 = [m_{i,j}^2]$, we have

$$\sigma^2 = M(2M_D - I) - M_2 \tag{3.5}$$

Hence

$$Var[X_{i,j}] = \sigma_{i,j}^2 \tag{3.6}$$

To represent the application as a DTMC, we consider its control flow graph. Assuming that an application has n components, with the initial component indexed by 1 and the final component by n, DTMC states represent the components and the transition from state i to state j represents the transfer of control from component i to component j. Following the procedure explained above, we can compute the expected number of visits to each component and its variance.

The DTMC representation, along with the concept of visit counts, has been used to express the system reliability as a function of component reliabilities.

In particular, denoting with R_i the reliability of component i, the system reliability is the product of individual reliabilities raised to the power of the number of visits to each component, denoted by $X_{1,i}$ (i.e., each component reliability is multiplied by itself as many times as the number of times it is visited starting from the first component); i.e., $R \approx \prod_i^n R_i^{X_{1,i}}$. Since the number of visits to a component is a random variable (except for the last component), the so-computed system reliability is also a random variable. Thus, denoting with E[R] the total expected reliability of the system, we have:

$$E[R] \approx \prod_{i}^{n} E[R_i^{X_{1,i}}] \approx (\prod_{i}^{n-1} R_i^{E[X_{1,i}]})R_n \tag{3.7}$$

where E$[X_{1,i}]$ is the expected number of visits to component i (and $X_{1,i}$ is always 1 for the final component n). The adopted model is known to belong to the class of hierarchical approaches; this kind of models, though approximate, lead to quicker and more tractable solutions than composite models [43]. To also take into account the second-order architectural effects and obtain a more accurate result, we can expand the above equation according to the Taylor series:

$$E[R] = [\prod_{i}^{n-1}(R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2\sigma_{1,i}^2)]R_n \tag{3.8}$$

Where $m_{1,i} =$ E$[X_{1,i}]$ and $\sigma_{1,i}^2 =$ Var$[X_{1,i}]$ (since $X_{1,n}$ is always 1, $m_{1,n} = 1$ and $\sigma_{1,n}^2 = 0$).

The second-order architectural effects are captured by the variance of the number of visits. The only source of approximation is the Taylor series cut-off. Note that the described model, as most of the architecture-based models, assumes independent failures among components. A more complex analysis would need to also consider failure dependencies. To complete the architectural model description, we add a state representing the Operating System (OS). The OS is considered as a component accessed through the system call interface. Transitions to the Operating System represent the transfer of control from components to the OS, i.e., a service request carried out via a system call.

Assume for now that the OS reliability per request is known and denote it with K (further details about how to estimate this value are in Section 3.4); we have that

$$K' = E[K^{X_{1,OS}}] = (K^{m_{1,OS}} + \frac{1}{2}(K^{m_{1,OS}})(\log K)^2\sigma^2_{1,OS}) \tag{3.9}$$

where $m_{1,OS}$ and $\sigma^2_{1,OS}$ respectively denote the mean and the variance in the number of visits to the OS. In the same manner, to simplify the notation, define:

$$R'_i = E[R_i^{X_{1,i}}] = (R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2\sigma^2_{1,i}) \tag{3.10}$$

and the expression 3.8 becomes:

$$E[R] = [\prod_i^{n-1} R'_i]R_n * K' \tag{3.11}$$

In the optimization problem, E[R] is required to be greater than a predefined level, R_{MIN} (R_{MIN} is an input) and R_i values are the decision variables (i.e., they are the output), while K is a given constant.

3.3 Optimization Model

In this Section we present the optimization model starting from a basic form and then enrich it by adding several features. The goal of the optimization model is to find the best combination of testing efforts to be devoted to each component so that they achieve a reliability level that can assure an overall reliability E[R] $\geq R_{MIN}$. Based on this model output, the tester will perform the verification activities focusing greater efforts on more critical components. If we assume that the reliability of each component grows with the testing time devoted to it, we can describe this relation by a software reliability growth model (SRGM), introduced in the Section 1.2.3. Such models are usually calibrated using failure data collected during testing; they are then employed for predictions, in order to answer questions such as “how long to test a software”, or “how many faults likely remained in the software”, and so on. In the context of SRGMs, the term failure intensity refers to the number of failures encountered per unit time; its form determines a wide variety of SRGMs. The time dimension over which reliability is assessed to grow, can be expressed as calendar time, clock time, CPU execution time, number of test-runs, or some similar measures. However, in general, the testing effort and its effectiveness do not vary linearly with time. The functions that describe how an effort is distributed over the exposure period, and how effective it is, are referred to as testing-effort functions (TEF). To address this issue, some SRGMs also include a TEF (Testing Effort Function) to describe this relation.

The model that we propose uses SRGMs, one for each component, to describe the relation between the reliability of a component and the testing effort devoted to it. This relation can be represented as $T = f(\lambda)$, where T is the Testing Time and λ is the failure intensity. Without loss of generality, we can consider “testing time” in place of “testing effort” ($T = f(\lambda)$): in fact, if the user of the model wants to consider the testing effort variation (for one or more components), s/he simply chooses an SRGM for that component that includes a Testing Effort Function (TEF) (like the cited ones) and put it in the model like any other SRGM. Hence, in the following, we refer to testing time and testing effort synonymously. This relation can be represented as $T = f(\lambda)$, where T is the Testing Time and λ is the failure intensity.

The general optimization model will then look like:

determine the optimal values of $T_1, T_2, \ldots, T_n$ so as to

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f_i(\lambda_i) \tag{12.a}$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R'_i] R_n * K' \geq R_{MIN} \tag{12.b}$$

with $i = 1 \ldots$ *n-1*, n components and T indicating the total testing time for the application to get a total reliability E[R] $\geq R_{MIN}$. Here λ_i variables are the decision variables (which determine the T_i variables and that are of course present in the constraint factors, via $R_i = \exp[-\int_0^{t_i} \lambda_i(\theta) d\theta])$.

Note that after the application has been tested according to the output of the model and then released, the component failure intensities are assumed to be constant; this is reasonable if the application developer does not debug or change the component during the operational phase.[1] With this assumption, the reliability of the component i at the end of the testing will be:

$$R_i = \exp[-\int_0^{t_i} \lambda_i(\theta) d\theta] = \exp[-\lambda_i t_i] \tag{3.13}$$

with t_i is the expected execution time per visit to component i; **this equation relates the failure intensity of component i to its reliability**. Each component can be characterized by a different SRGM (among the plethora of proposed ones). For instance, if we assume the Goel-Okumoto model for all the components (for which $\lambda(T_i) = a_i g_i e^{-g_i T_i}$),

[1] This assumption, anyway, is not that important for us, because the problem of estimating the reliability variation during operation or maintenance is a different problem; for our purpose, i.e, satisfying the reliability constraint at the software release (i.e., at the end of testing phase) in minimal tesitng time, the stated assumption is not that relevant.

the objective function becomes T = $\sum_i (1/g_i ln(a_i g_i/\lambda_i))$, where a_i is the expected number of initial faults, λ_i is the desired failure intensity and g_i is the decay parameter. Other SRGMs can be used to represent the *reliability-testing time* relation for each component. This general model can be specialized according to the information the tester has, in order to get different accuracy levels in the solution in a flexible way. Assume that the tester has no knowledge about the components and their previous history. In other words, s/he does not have any historical data related to previous testing campaigns performed on the components (or on similar ones), and s/he is not able to obtain such information from the current version. Without such knowledge, s/he cannot build the SRGMs for these. In this case a basic, minimal solution can be obtained. In particular, the output will be the reliability that each component needs to achieve, assuming that the testing time to achieve a failure intensity λ is the same for all components, as though they were described by the same SRGM with identical parameters (i.e., $f_i(\lambda_i) = f(\lambda_i)$). In this case, the model will then look like:

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f_i(\lambda_i) = \sum_{i=1}^{n} f(\lambda_i) \tag{14.a}$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R_i']R_n * K' \geq R_{MIN} \tag{14.b}$$

In other words, in this case the model does not predict the testing times needed to achieve the required reliability. Therefore, the results have to be interpreted as an indication of the most critical components in the architecture or, equivalently, as the reliability values each component needs to achieve to satisfy E[R] = R_{MIN}. Measuring the reliability during the testing, (e.g. as in [113]), the engineers will know when the testing for each component can be stopped. We call this solution the ***basic solution***.

If some qualitative indications about the testing cost of components are available, it is possible to include them as weights in the objective function. For instance, information about process/product metrics, that is easily obtainable, can be used to estimate their fault content. Regression trees are very useful for this purpose, but also the simpler fault density approach can be used. By including weights proportional to the estimated fault content, the model solution not only gives the reliabilities the components need to achieve, but also an indication about the *relative* testing efforts to make them achieve such reliabilities, according to the components complexity. The previous assumption is relaxed and becomes: each component needs an amount of testing time that is proportional to its fault content, and hence, indirectly to its complexity. A possible way to include the weights in the objective function is the following:

$$T_i = f(\frac{\lambda_i}{1 + rWeight_i}) \tag{3.15}$$

leading to the following model:

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f(\frac{\lambda_i}{1 + rWeight_i}) \tag{16.a}$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R'_i] R_n * K' \geq R_{MIN} \tag{16.b}$$

where $f(\lambda_i)$ is a default SRGM, the same for every component with the same parameter as in the basic solution, and $rWeight_i$ (i.e., reliability weights) is given by the proportion of the fault content in the component i with respect to the entire system's estimated fault content (i.e. $faultContent_i$ / $\sum_i faultContent_i$). We call this solution ***extended solution***. Note that in order for the weights to really increase the needed testing time when they increase, the f(λ) function (the inverse of the default SRGM) is preferable to be a decreasing function, rather than an increasing/decreasing function (as for instance, the one derived by the Goel-Okumoto SRGM), and a simple function to be evaluated. The adoption of a non-decreasing function would require some expedients to be adopted for it to work correctly; however, since the only important requirement is that this function be the same for all the components, it makes no sense choosing a complex function. The most appropriate choice is a simple SRGM, like the Goel-Okumoto model.

The general model described by the equation 12 can be fully exploited when engineers also collect information about the failure behavior of the components. In this case the optimal testing times needed for each component can be predicted. In fact, data about the components failure behavior allows engineers to build a reliability growth model for each one of the component.

Engineers often collect this kind of information during the testing process for various purposes, like improving the process, assessing the achieved reliability after some testing time, building reliability growth models, and use it in the same or in successive projects for scheduling optimal release policies.

In this case, failure data, (i.e., interfailure times or the fault density along with the coverage function [43]) are used to fit the best SRGM (the issue of how to fit an SRGM for the current project is described in Section 3.4). By solving the model with the SRGMs, the *absolute* testing times to be devoted to each component are obtained. The accuracy of the result depends on how well the testing processes of the components are described by their corresponding SRGMs. The previous assumptions are replaced by the common SRGM assumptions (as summarized in [115] and many other related papers). We call this general solution, epressed by equation 12, ***complete solution***.

Finally, note that the model expression 12 is generalized to the case of multiple applications by adding a reliability expression, of the form of equation 3.11, for each application as a constraint. The solution for multiple-applications problems usually requires heuristic approaches (a genetic algorithm in our case).

3.3.1 Performance Testing Time Contribution

During the testing process of a critical application, part of testing resources could be reserved to tune the application performance in order to fulfill performance requirements. This is especially true for real-time systems. To also consider the additional testing resources each component can require for performance testing we should know the relation between the performance improvement and the performance testing time (i.e., a sort of *performance growth model*). Even if this could be a topic of future research, more realistically it is difficult to infer such relations, differently from reliability growth models. For this reason, it is easier to include the performance testing time contributions as weights in the objective function. In the extended solution case, where no SRGM was available, we saw that the *rWeight* values represent the assumed proportionality between the testing time and the estimated fault content (and indirectly to process/product metrics). In a similar way, the performance weights will represent the proportionality between the performance testing time and some performance metric, e.g., the expected total execution time for a component (ETET). Denoting with t_i the expected execution time per visit for the component i, the $ETET_i$ is given by $ETET_i = V_i t_i$. The $argmax_i\{V_i t_i\}$ is the performance bottleneck of the application [43]. Thus, assuming that performance testing time will be proportionally devoted to components according to their $ETET_i$ value, we can add a weight, named *pWeight*, in the objective function computed as $(ETET_i \; / \sum_i ETET_i)$. This weight represents the *performance testing time contribution.* In this way, performance bottlenecks will receive more performance testing time. A possible objective function for the extended solution (and the basic one, if $rWeight_i = 0$) can be the following:

$$\text{Minimize} \;\; T = \sum_{i=1}^{n} (f(\frac{\lambda_i}{1 + rWeight_i}) * [1 + PF * pWeight_i]) \tag{3.17}$$

where the first term in the sum has the same meaning as in equation 16 and PF is a factor between 0 and 1 representing the percentage of time the tester wants to devote to performance tests. When more information is available (like maximum tolerable execution times for each component, as in the case of real time systems) different weights are possible (e.g., ranking the component based on the difference between the total expected execution time and the imposed maximum execution time; the greater is the difference, the more performance testing time they need). For the complete solution, when the objective function is the sum of the testing time functions $f_i(\lambda_i)$ (i.e. the inverse of SRGMs), the performance weights, $pWeight$, can be added in the same way, as a penalty, due to the "performance testing" time to be devoted to each component. We thus have:

$$\text{Minimize} \quad T = \sum_{i=1}^{n}(f_i(\lambda_i) * [1 + PF * pWeight_i]) \tag{3.18}$$

In both cases, accounting for performance testing time will be optional (if the tester does not want to account for it, PF will be 0). In the future, we plan to explore the relation among performance testing times and performance improvement, in order to formulate a testing resource allocation problem, with both minimum reliability and minimum performance levels as constraints.

3.3.2 Fault Tolerance Mechanisms

The model considers the potential means of failure mitigation that one or more components could employ. Main mechanisms that are adopted in critical systems, namely, the restart-component, retry-application and failover to a standby are considered here. Denote as "subsystem C" a component along with its standby version (in the case of no standby version, C denotes a single component). The fault tolerance mechanisms are considered in the following order (Figure 3.1): if a failure occurs, and the failure is detected, the first recovery attempt is to restart-component operation; the second recovery attempt, if the first one fails, is to retry-application and the final operation is a failover to a standby version, once the other actions have failed. The expression we derive below for the reliability of such a subsystem can describe components implementing one or more of these fault tolerance mechanisms. Other mechanisms can easily be added to the model, by deriving the corresponding reliability expression.

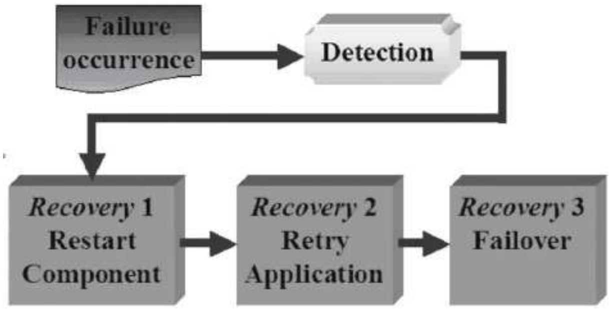


Figure 3.1: Fault Tolerance Mechanisms

In this case, we need to consider the following events in order to derive an expression for the reliability of the subsystem C:

- **DETFailed** = detection failed given a failure occurred
- **RESFailed** = the restart failed given a failure and detection occurred
- **RETFailed** = the retry failed, given a failure occurred, detection occurred and the restart failed
- **FOFailed** = the failover failed given a failure occurred, detection occurred, the restart failed and the retry also failed.

Describing the component reliabilities as in the equation 3.13, the failure probability of a component will be F_i = 1 - R_i = 1 - exp[-λt] , i.e. the time-to-failure (TTF) has an exponential distribution. When a failure occurs in the primary component of a subsystem C, the following group of events (we denote them as E1, E2, E3) can take place:

1. The detection fails, or the detection succeeds but the restart, retry and failover operations subsequently fail (i.e. $P_{E_1} = P_{DET_{Failed}} + (1$ - $P_{DET_{Failed}})(P_{RES_{Failed}} P_{RET_{Failed}} P_{FO_{Failed}})$: in this case the failure has not been covered and the conditional TTF distribution of the subsystem C is given by the TTF of the primary version:
$$EXP(\lambda) = 1 - e^{-\lambda t} \tag{3.19}$$
2. The detection succeeds and the restart also succeeds, or the retry succeeds after the restart failed (i.e. $P_{E_2} = (1$ - $P_{DET_{Failed}})[(1$ - $P_{RES_{Failed}})$+$P_{RES_{Failed}}(1$ - $P_{RET_{Failed}})])$. In this case, the failure has been covered and the same component starts operating from scratch. Thus, the conditional TTF distribution is given by the sequence of two identical exponentially distribution representing the same component rerun, i.e., by a 2-stage *Erlang*:
$$ERLANG(\lambda, 2) = 1 - (e^{-\lambda t}(1 + \lambda t))) \tag{3.20}$$
3. The detection succeeds, the restart/retry operations fail and the failover to a standby version succeeds (i.e.P_{E_3} = $(1$ - $P_{DET_{Failed}})[P_{RES_{Failed}} P_{RET_{Failed}}(1$ - $P_{FO_{Failed}})]$). Thus, after a failure occurrence, the standby version will be activated. When this event occurs, if the standby is an identical copy of the primary version, the conditional TTF distribution is given again by the equation 3.20. If a different version is used, the distribution of the conditional TTF is given by the sequence of two independent exponential distributions (describing the primary and the standby TTF), which

is known to be a Hypoexponential distribution. Given λ and λ_1, the failure intensities respectively of the primary version and its standby, the TTF distribution will be

$$HYPO(\lambda, \lambda_1) = 1 - (\frac{\lambda_1}{\lambda_1 - \lambda})e^{-\lambda t} + (\frac{\lambda}{\lambda_1 - \lambda})e^{-\lambda_1 t} \tag{3.21}$$

The most general reliability expression, for a component that owns all of the considered mitigation means, will be:

$$\begin{gathered} R_C = 1 - [P_{E_1} * EXP(\lambda) + P_{E_2} * ERLANG(\lambda, 2) + \\ + P_{E_3} * (sameVersion * ERLANG(\lambda, 2) + \\ + !sameVersion * HYPO(\lambda, \lambda_1))] \end{gathered} \tag{3.22}$$

where *sameVersion* is 1 if the standby is identical to the primary copy, 0 otherwise. If a component does not have any mitigation means, the probabilities $P_{RES_{Failed}}$, $P_{RET_{Failed}}$ $P_{FO_{Failed}}$ will be 1 and P_{E_1} will also be 1, while P_{E_2} and P_{E_3} will be 0:

$$R_C = 1 - EXP(\lambda) \tag{3.23}$$

If a component has only the restart/retry mechanism and not a standby version, the $P_{FO_Failed} = 1$, $P_{E_3} = 0$ and the expression becomes:

$$R_C = 1 - [EXP(\lambda)P_{E_1} + ERLANG(\lambda, 2)P_{E_2}] \tag{3.24}$$

Finally, if a component has a stand-by version, but not the restart/retry mechanism, $P_{RES_{Failed}}$ and $P_{RET_{Failed}}$ are 1, $P_{E_2} = 0$ and the expression is:

$$\begin{gathered} R_C = 1 - [P_{E_1} * EXP(\lambda) + \\ + P_{E_3} * (sameVersion * ERLANG(\lambda, 2) + \\ + sameVersion * HYPO(\lambda, \lambda_1))] \end{gathered} \tag{3.25}$$

By replacing R_i by the R_{C_i} expression in the equation 3.10, the model will be described by the following:

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} (f_i(\lambda_i)) \tag{3.26}$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R'_{C_i}]R_{C_n} * K' \geq R_{MIN}$$

This general form allows adding any other failure mitigation means for the component i, by finding the corresponding expression for R_{C_i}. Similarly, if one wants to adopt more complex expressions to describe the reliability of a subsystem C, it is sufficent to replace the discussed R_{C_i} expressions with the new expression (for instance, in the discussed expression for the failover to a standby version, the statistical independence is assumed; the R_{C_i} expression can be replaced by more complex expressions accounting for any form of dependence). The parameters estimation ($P_{DET_{Failed}}$, $P_{RES_{Failed}}$, $P_{RET_{Failed}}$, $P_{FO_{Failed}}$) is briefly discussed in the next Section.

3.4 Information Extraction

The described approach is designed to provide different levels of solution according to the available information. To obtain a minimal solution, the basic information to be provided is related to the architecture (components identification, transition probabilities), and each component (expected execution time per visit (t) or expected total execution time (ETET) and the OS reliability). Additional information about the component fault density and some process/products metrics allows for computing the extended solution. Further information about the interfailure times, or the *coverage testing function / faults contents* for the components allows to obtain the SRGMs, and thus the complete solution. The ETET (or equivalently t) for the components also allows the performance testing time to be included in the solution. Finally, to include one of the described fault tolerance mechanisms, the corresponding parameters need to be estimated.

Since the model results have to be applied in the testing phase, there are basically two different ways to obtain such information: by design/code information and simulation before the testing or by dynamically profiling a real execution from system test cases of a previous version. The former refers to design documents (such as UML diagrams), and to static code analysis tools (mainly static profiling techniques and simulation tools). The latter refers to the execution of the system test cases, which emulate the system functionalities (thus, in this case a previous version of the software has to be profiled, since for the current version the testing still has not started). Dynamic profiling solution assumes that the executions will represent the real operational usage; this is generally accomplished in two ways: by assessing the operational profile and assigning an execution probability to each functionality or by shuffling and re-executing the system functional test cases, averaging the results.

Neither the design/code nor the dynamic profiling approach is the best one for all the parameters. An approach can be better for some parameters and worse for others.

The advantage of design/code-based estimations is that it relies on the current system version and not on a previous one. On the other hand, for some information (such as the ETET or transition probabilities) the dynamic profiling approach can be easier to use. As for the accuracy, if the system does not change much between the profiled version and the current version, execution traces would be more accurate than design-based approaches; but if the current version introduced significant changes in the code, a design-based approach would be better. If possible, a combination of both is probably the best solution: values obtained from past execution traces can be refined by reflecting the changes in the new version (which could have altered some values).

A third important way can be useful for both the approaches: expert judgments and historical data from similar systems. Basic information could be obtained as follows:

- **Architecture**: components (at each granularity level) are normally identifiable from design documents. When documents are not available, the architecture can be extracted by using some source code[2] as well as object code extraction tools. It can also be derived by traces resulting from a dynamic profiling tool, such as gprof[3]. Transition probabilities can also be derived by both approaches. As for the design phase, the estimation can be accomplished by a scenario-based approach [104], by simulation, by static profiling of the execution or by interviewing the program users, as in [113]. As for the dynamic profiling, transition probabilities can be estimated by counting the number of times the control passes from a component to others (or to itself): point estimate of the transition probability from component i to component j will be given by ($\mathit{NumberOfTransfers}_{i,j}$ / $\sum_j NumberOfTransfer_{i,j}$).

  In particular, from the output of a profiler tool, the flat profile and the call graph can be obtained: information provided by the flat profile (i.e., how much time a program spent in each function and how many times that function was called) and by the call graph (i.e. information regarding the other functions calling a particular function and the functions called by it) allows us to construct the DTMC model and also get the

[2] for instance, SWAGkit tool. Available from: http://www.swag.uwaterloo.ca/swag-1354kit/index.html.

[3] GNU gprof. Available from: www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

relevant transition probabilities. A possible way to get these parameters is outlined in [44]. Finally, since the OS is usually not subject to changes, its reliability can be estimated by dynamic profiling the previous version or also other similar systems using the same OS. The point estimate is $K = 1 - lim_n F/N$, where F is the number of observed OS failures and N is the number of test cases.

- **Components**: The ETET of a component (or equivalently t) in the design phase can be obtained as in the case of the transition probabilities. Simulation could be more suitable in this case. However, this parameter is more easily obtainable experimentally, by the profiling approach, since the time spent in a component is an output of many profiler tools. Note that the accuracy of the expected time per visit, t, also depends on the granularity level chosen for a visit: for instance, in [43] a visit is intended as the execution of a basic block[4] of instruction (that in average was 2 lines of code) and, as a result, the expected time per visit did not differ significantly among visits. This consideration also stands for the transition probabilities accuracy. For the extended solution, we need an estimation of the component fault contents, to compute $rWeight$ values. Following the fault density approach, we can estimate the fault density (FD) from past experience of the same system (previous versions) or similar systems, or by assuming common values from the literature or also by expert's judgment. The fault content of component i will be $\phi_i = FD * LOC_i$, where LOC is the number of

[4] A basic block, or simply a block, is a sequence of instructions that, except for the last instruction, is free of branches and function calls. The instructions in any basic block are either executed all together, or not at all.

code lines. Following the regression trees approach, we still need the estimated fault content of the system and some complexity metrics of the components to derive a more accurate estimation than the FD approach of the components' fault contents. Finally, for the complete solution, the data necessary to build the SRGMs can be the interfailure times or alternatively the testing coverage function along with the estimated component fault content. Both of these are obtainable by the dynamic profiling approach (of the previous as well as the current version) or by historical data from the same or similar components.

In particular, when a component does not undergo significant changes from version to version, the SRGM parameters estimation based on the collected failure data can be used; when a component is significantly changed in the new version, some parameters can still be built using the collected data and refined through the current version information, while others are completely to be estimated from the current version information (e.g., the initial fault content, a common parameter, can be estimated considering the complexity metrics of the new version, with the fault density approach or regression trees). Some recent studies [115], estimating the number of faults during the current version testing, reported that after about a 25% of the total testing time, several SRGMs (both finite and infinite NHPP models) prediction accuracy deviated by only 20%. For a testing resource allocation problem, this means that initially the computed resources allocation will be affected by SRGM errors, but dynamically re-computing the optimal allocation at some time intervals will give more and more accurate results. It is however worth to point out that SRGMs are known to give good results even when data partly violates the model's assumptions [115] they are based on, and their usage is therefore encouraged.

Similar approaches can be used to estimate the parameters for the fault tolerance mechanisms. Depending upon the actual mechanisms used, the failure probability of the failure detection, restart/retry or failover operations need to be estimated. In particular, some potential solutions are:

- Estimating the values from historical data form previous similar systems or common values assumed for the fault tolerance manager components in other critical systems or from the literature.
- Doing a fault injection campaign and obtain:
  - *#Failed detections / #faults injected*
  - *#Failed restarts / #attempted restarts*
  - *#Failed retries / #attempted retries*
  - *#Failed failover / #attempted failover*

It is finally worth to point out that for all the cited parameters expert judgment should not be neglected and should always be used to refine the other estimation method results.

In general, it is difficult to have a common method to assess the quality of the extracted information and to evaluate how much it impacts on results. However, we believe that the most effective way to do this, is to evaluate the impact of the information quality on the specific model where it is used, by performing a sensitivity analysis on all the estimated parameters (as we did in the case study). This method is at once practical and accurate. It allows us to understand and balance possible estimation errors of input parameters (caused by the low quality of extracted information) and their effects on the solution.

3.5 Experimental Evaluation

The case study chosen for illustrative purpose is an application, written in C, developed for the European Space Agency. It is a program to provide user interface for the configuration of an array of antennas. The program consists of about 10000 lines of code. Its purpose is to prepare a data file according to a predefined format and characteristics from a user, given the array antenna configuration described using an appropriate Array Definition Language. The program is divided into three main subsystems: the Parser module, the Computational module and the Formatting module.

This case study has been used in other studies about reliability analysis and evaluation (e.g., in [113],[44]),

Figure 3.2 shows the architectural model of the system.

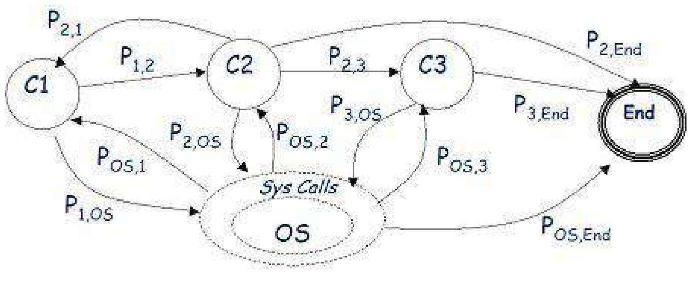


Figure 3.2: Software Architecture

The granularity of component (i.e., the level of decomposition), is an analysis choice that depends on the needs. In general, few large components results in easier computational analyses and in a greater amount of available data that allow us to build an accurate SRGM (particularly useful in the case of *complete solution*). The choice of component granularity also depends on how much they are decoupled. High decoupling indicates that components can be more independently tested than with coupled components: this enables the possibility to better schedule testing activities and organize the testing team to work on different components, by using results obtained with the allocation model.

The granularity of component, in this case, is chosen to be a subsystem. No fault tolerance mechanism, as those described above, is present in this system. All the described kinds of solution (i.e., basic, extended and complete, with or without performance, with or without fault tolerance means, single or multiple applications) can be obtained with the implemented tool, depending on the features of the application under study. In the conducted experiments we computed a single-application complete solution, with performance and without fault tolerance means. To estimate the parameters we used a hybrid approach, exploiting dynamic profiling and design/code information. In particular, the experimental procedure is outlined in the following basic steps:

1. Creation of a faulty version of the program, by reinserting faults belonging to real fault set discovered during integration testing and operational usage (Table 3.1). This faulty version emulates the previous version of the application. Note that it is likely that the version of the application we used contains very few faults (except the ones inserted by us), since it has been extensively used without having failures for a long

time.

Table 3.1: Types of injected faults

Fault Types and Subtypes	#Injected Faults	Type Number
Logic omitted or incorrect		
Forgotten cases or steps	4	*1*
Unnecessary Functions	2	*2*
Missing Condition Test	10	*3*
Checking Wrong Variable	4	*4*
Computational Problems		
Equation insufficient or incorrect	20	*5*
Interface incorrect or incomplete		
Module mismatch	6	*6*
Data Handling Problems		
Data initialized incorrectly	4	*7*
Data accessed or stored incorrectly	20	*8*
Total	**70**	

2. Testing execution for the faulty version for a certain amount of total testing time. Only a fraction of the injected faults are removed. During the tests of this "previous version", the application is profiled. From execution traces the DTMC model and the transition probabilities are obtained. The OS is included among the components. Failure data and execution times are also collected during this phase.

3. Based on the collected failure data, an SRGM for each component is determined. We used SREPT functionalities to obtain the SRGMs.

4. Applying the optimization model, testing times for each component is predicted for the current version. Our tool uses the sequential quadratic programming algorithm to solve the non-linear constrained optimization problem. The current version is then

tested according to the computed test times allocation.

5. At the end of the testing, the reliability predicted by the model is compared with the actual achieved reliability, computed as by $(1 - lim_n Nf/N)$, where Nf is the number of observed failures and N is the number of executions of input cases. The prediction error is then analyzed against the possible prediction errors that could occur in the (i) transition probabilities estimation, (ii) in the SRGMs and (iii) in the OS reliability estimation, by carrying out a sensitivity analysis.

3.5.1 Results and Analysis

Step 1

According to the described steps, we injected 70 faults in the software (31, 28 and 11 respectively in the component 1, 2 and 3) based on the fault categories of Table 3.1. An excerpt of the injected faults with the corresponding detection testing time and test case number is reported in Table 3.2.

Step 2

Test execution for this faulty version was carried out by randomly generating test cases based on the operational profile (in this phase, 366 test cases were generated). After the testing phase, 46 faults were removed (respectively 20, 19 and 7), leaving 24 faults in the software. The reliability of this first version of the software was measured executing further 3600 test cases (picked up from the operational profile) and recording the number of failures, without removing the corresponding faults. It amounted to $R = 1 - lim_n Nf/N = 0.93583$,

Table 3.2: An excerpt of the Injected Faults

Fault Number	Function	Line	Type	Component	Detect. Time	Test Case
2	*Nodedef*	62	8	1	0.124	2
3	*Hexdef*	100	8	1	0.949	7
4	*Nodecoor*	49	8	1	1.074	8
...	...	...	...	...	...	...
37	*Fixsgrel*	99	4	2	2.121	17
38	*Fixsgrpha*	29	4	2	2.414	20
39	*Fixsgrid*	64	4	2	2.987	24
...	...	...	...	...	...	...
68	*Gwrite*	110	7	3	14.412	120

with $Nf = 231$ and $N = 3600$. We assumed a reliability goal for the next release of R_{MIN} = 0.99; thus the testing resources for the current version are to be allocated according to this goal. We profiled the previous test executions by gprof[5] (for the user functions) and by straceNT[6] (for the system calls), obtaining the execution counts among the components, the corresponding transition probabilities (as described in the previous section) and then the visit counts.

In particular, a visit to an application component is a flow of the control to a user function coming either from a caller user function or from a return by a called function (user function or system call). Thus, a user function F calling another function will have two visits: one from the caller function and another from the return of the called function. The average time per visit to application components during an execution can be computed using the

[5] MinGW (Minimalist GNU for windows) has been used to provide *gprof* and other GNU tools under windows. See http://www.mingw.org/.

[6] straceNT is a system call tracer for Windows See http://www.intellectualheaven.com/.

following formula:

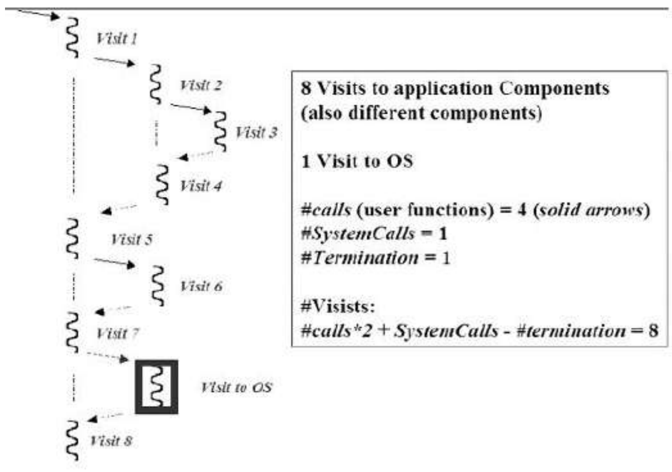
$$\begin{gathered} TV = totalTime/(2 * \#calls + systemCalls - \\ \#termination * averageDepth) \end{gathered} \tag{3.27}$$

where *totalTime* is the average total user function time per execution (an output of *gprof*, averaged over test case executions), and the denominator is the average number of visits per execution to all the application components: *#calls* is the number of user functions called per execution (doubled to consider the return), *systemCalls* is the number of calls to the OS (an output of *sTrace*), not doubled because we are counting the visits to the application components, not also to the OS; *#termination* is the number of terminations (either normal or abnormal) and *averageDepth* is the average depth of the call graph, included in order to subtract the "returns" from a function that are lost due to the termination (this value has been computed by analyzing the output of *gprof*, and was equal to 2.6). Figure 3.3 clarifies this computation with an example (with a normal terminating single execution and *averageDepth* = 1). Execution counts for a component are computed similarly to the denominator of equation 3.27. The difference is that only the calls from a component to itself are doubled, in order to consider the return of the control flow to itself. Calls to the other components (and to the OS) are not doubled, because in that case just the "return" from the called component has to be accounted as a visit to the calling one.

However, for a better accuracy, we actually estimated the time per visit for each one of the components, considering:

$$TV_i = totalTime_i/visitCounts_i \tag{3.28}$$

where *totalTime*$_i$ is the proportion of user time spent in the component i (computed as *totalTime* * #*calls*$_i$ / #*calls*) and *visitCounts*$_i$ are computed from transition probabilities (in turn derived from execution counts) with the procedure described in the Section 3.2. Results are summarized in Table 3.3. As for the OS, the visit granularity is slightly different. Since OS was proprietary, it was not possible to trace the internal function calls (and this was the reason why we did not consider the block as visit granularity); thus the visit in this case is the flow of the control coming from a user function to the OS via a system call; it does not consider the control flow from the OS to itself through internal kernel function calls. Correspondingly, the time per visit will also have a rougher granularity: it is the average time spent in an entire system call execution, computed as the average OS time (obtained by timeit[7]) divided by the average number of system calls (i.e., the average



8 Visits to application Components (also different components)

1 Visit to OS

***#calls* (user functions) = 4 (*solid arrows*)**
***#SystemCalls* = 1**
***#Termination* = 1**

#Visists:
***#calls*2 + SystemCalls - #termination* = 8**

Figure 3.3: Granularity of Visit

[7] *timeit* is a command-line tool, provided with Microsoft Windows Resource Kit Tools, that records the time a specified command takes to run.

number of visits in this case). Their product is always the total time spent in the OS.

The return of the system call is accounted as a visit for the calling component from the OS, as explained above.

Their values (execution counts/number of visits and times per visit), however, are not of interest for the OS reliability itself, because the OS reliability (i) is given in this case for the entire execution (not as reliability per visit) and (ii) it is assumed constant with time. But the execution counts to and from the OS are important to determine the visit counts for other components.

Also the variance of the visit counts is considered, for the second-order architectural effects. Finally the performance factor is set to 0.1, i.e., only the 10% of testing efforts will be employed for performance testing. Results of this step are summarized in Table 3.3.

Note that since the visit granularity is so fine, the expected visit counts during an execution are very high and the transition probabilities toward the end state are very low (because only a minimal part of code leads to the end). Moreover, also note that the Parser subsystem (component 1) and the Formatting subsystem (component 3) make a great use of the OS, and the corresponding transition probabilities are significantly higher than the Computational subsystem.

The high values for visit counts could also be due to the first-order DTMC: a first-order DTMC does not allow one to consider the dependence of transition probability from a component i to j on the current as well as the previous components from which the control arrived at component i. However, we also considered the second-order DTMC, detecting

Table 3.3: Estimated Paramter Values

<table>
<tr><td colspan="5"><b>Minimum Required Reliability</b></td><td>0.99</td></tr>
<tr><td colspan="5"><b>Reliability of the Previous Version</b></td><td>0.93583</td></tr>
<tr><td colspan="5"><b>Mean User Process Execution Time</b></td><td>0.01389</td></tr>
<tr><td colspan="5"><b>Mean OS Execution Time</b></td><td>0.10589</td></tr>
<tr><td colspan="5"><b>Mean # User Calls</b></td><td>407.4</td></tr>
<tr><td colspan="5"><b>Mean # System Calls</b></td><td>5442</td></tr>
<tr><td colspan="6"><b>Transition Probabilities</b></td></tr>
<tr><td>TO
FROM</td><td><b>1</b></td><td><b>2</b></td><td><b>3</b></td><td><b>OS</b></td><td><b>End</b></td></tr>
<tr><td><b>1</b></td><td>0.2307</td><td>2.71E-4</td><td>0</td><td>0.7689</td><td>7.79E-5</td></tr>
<tr><td><b>2</b></td><td>0</td><td>0.65253</td><td>0.0298</td><td>0.3439</td><td>4.96E-4</td></tr>
<tr><td><b>3</b></td><td>0</td><td>0</td><td>7.72E-4</td><td>0.99902</td><td>2.10E-4</td></tr>
<tr><td><b>OS</b></td><td>0.4049</td><td>0.0141</td><td>0.5810</td><td>0</td><td>0</td></tr>
<tr><td><b>END</b></td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td><b>Visit Counts</b></td><td>2865.8</td><td>222.8</td><td>3165.0</td><td>5442.3</td><td>-</td></tr>
<tr><td><b>Exec. Time</b></td><td>0.01128</td><td>0.00248</td><td>1.251E-4</td><td>0.10589</td><td>-</td></tr>
<tr><td colspan="6"><b>SRGM</b>: Exponential SRGM. λ(t) = age^{-gt}</td></tr>
<tr><td colspan="2"><b>1</b></td><td colspan="2"><b>2</b></td><td colspan="2"><b>3</b></td></tr>
<tr><td><i>a</i></td><td><i>g</i></td><td><i>a</i></td><td><i>g</i></td><td><i>a</i></td><td><i>g</i></td></tr>
<tr><td>13</td><td>5.46E-2</td><td>11</td><td>9.46E-2</td><td>5</td><td>5.34E-2</td></tr>
</table>

no significant changes in the transition probabilities. Thus to keep the treatment simple, the first-order results are considered in the following. Finally, we did not observe failures due to the OS; thus we estimate an OS reliability value equals to 1. This value is clearly an overestimation due to the low number of test cases used to estimate it; thus we start with this value, but it will be varied (*step 5*) between 0.995 and 1.0 to take into account the possible estimation error (this will also show how the overall reliability estimate is affected by the OS reliability value).

Step 3

Based on the interfailure times (Table 3.2) of the previous version testing, an SRGM for each component was built, using SREPT to fit the best model to the data. The fault content parameters for the current version were derived from the estimated remaining fault contents, while the rate parameters were set at the same value of the previous testing process (Table 3.3).

For all of the components, the same kind of SRGM was found to be the best fitting one (i.e., the Goel-Okumoto model): this is mainly due to the strong similarities among the testing processes followed for the three components.

Step 4

With the visit count values, the SRGMs and the OS reliability, the model was built and solved by our tool, giving as output the optimal testing times for each component.

After executing the tests (always by generating test cases from the operational profile) according to the optimal testing times, 19 faults of 24 were removed (respectively 11 of 11, 5 of 9 and 3 of 4). Table 3.4 shows the testing times devoted to each component, the corresponding number of executed test cases, the **detection time** and the detecting test case number for each fault. (Clearly the actual devoted time will slightly exceed the allotted time, because the latter is not a perfect multiple of the execution time of a test case).

Step 5

According to the model, the final reliability should be 0.990289. Measuring the actual reliability using the same procedure used for the previous version, it turned out to be 0.989722, with 37 observed failures over 3600 executions. The relative error is about 5.7289E-4 resulting in an overestimation of 0.057289 %.

The main sources of error in the prediction are i) the DTMC transition probabilities and

Table 3.4: Testing of the system according to the model results

Component	#Fault Removed	Optimal Testing Time	#Test Cases
1	11	40.638	348
2	5	6.315	58
3	3	24.682	215
Fault Number		Detection Time	Test Case Number
Component 1			
21		1.213	10
22		3.112	25
23		6.452	53
24		6.992	57
25		8.346	69
26		12.240	102
27		13.332	111
28		15.341	128
29		22.021	183
30		30.041	250
31		38.098 of 40.638	318 of 348
Component 2			
51		0.933	7
52		2.729	22
53		2.981	24
54		4.065	33
55		6.209 of 6.315	51 of 58
Component 3			
69		5.034	42
70		7.355	61
71		20.442 of 24.682	170 of 215

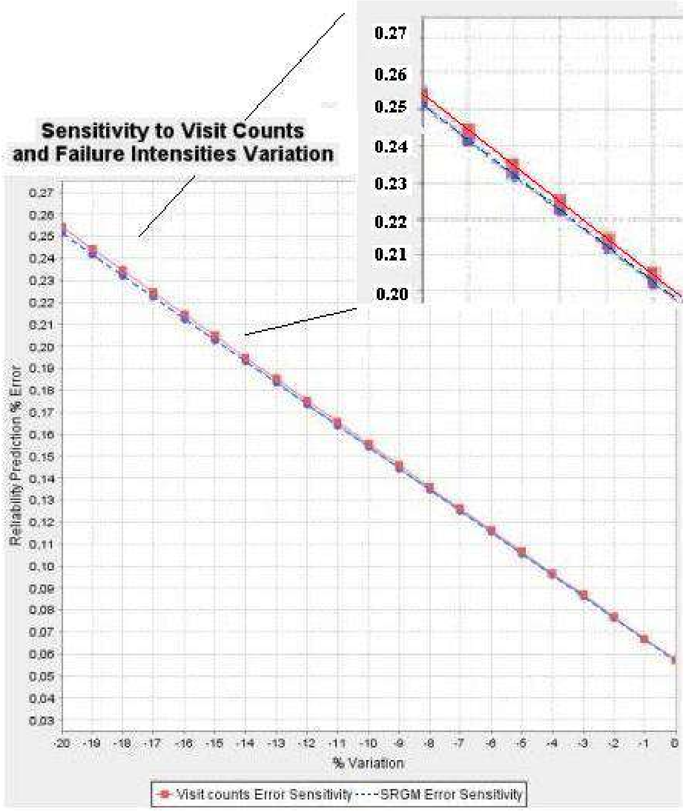


Figure 3.4: Sensitivity to visit counts and failure intensities variation

consequently the visit counts, ii) the SRGMs prediction ability and (iii) the OS reliability. In our case the architecture of the application and the components themselves have not significantly changed as well as the testing process between the two versions. Such a good prediction is also due to this. However, we can see how in the presence of significant changes in such values, the prediction is still good. First, figure 3.4 shows the variation of the percentage relative error in the reliability prediction against the variation of the percentage relative error in all the visit counts (solid line). For a maximum of 20% in the visit counts estimation errors (underestimation) we have a reliability prediction error of about 0.257%

, i.e., a prediction of about 0.99226.

Second, in the same figure (figure 3.4, dashed line) the same variation in the prediction against the percentage relative error in the failure intensities is shown: if an SRGM does not accurately describe the testing process, the value of the achieved failure intensities at the end of the testing time for each component will be affected. We evaluated such effect by varying the failure intensities. Results in figure 3.4 show that the reliability prediction will be affected by an error of 0.256% , i.e. 0.99225, for a percentage variation of 20% in the failure intensities (underestimation). Failure intensities plot shows almost the same behavior as the visit counts plot, because in the reliability computation their product appear in the exponents (the slight difference is due to the second-order part in the formula).

Third, since no OS failures have been observed, we estimated the OS reliability to be equal to 1; figure 3.5 shows the effect of an estimation error in the OS reliability on the overall reliability prediction. In this case the overall prediction is more sensitive to OS reliability prediction errors than the previous parameters.

However, the previous percentage errors simply do not make sense in this case; it is very unlikely to mistake the OS reliability prediction with an error of 20%. For instance, suppose that in our case an OS failure over 3600 execution tests has been observed and we estimate the reliability as (1-1/3600) = 0.9997; an error of 20% would mean making an estimation of 0.79976, that in our case would correspond to about 720 failures: i.e., in the current version the OS has experiences 719 failures more than the previous version over only 3600 test cases. It is a huge prediction mistake. Moreover, OS has usually much more historical failure data than the other application components; this makes OS reliability estimation easier and more accurate. Figure 3.5 shows therefore the percentage error variation of the overall reliability depending on the absolute variation of OS reliability in a reasonable range.

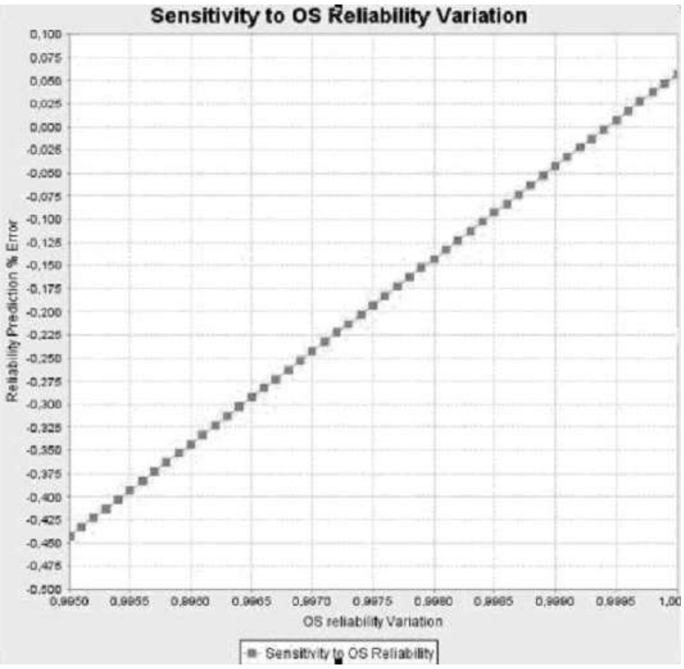


Figure 3.5: Sensitivity to OS Reliability variation

In this case, for a maximum error of 0.005 (i.e., the 0.5%), the prediction error shifts from the original overestimation of 0.057289% with respect to the actual reliability to an underestimation of about 0.443% , i.e., a reliability of 0.98534. The solution is therefore much more sensitive to the OS reliability estimation than the other parameters.

Finally, all the results have been obtained with the estimated operational profile; changing the operational profile will yield different results. The effect of the operational profile variation on reliability measurements (particularly on SRGMs) has been studied by Pasquini et al. in [113], which however pointed out that the “predictive accuracy of the models is not heavily affected by errors in the OP-estimate. In general, if a model-estimate fits the actual reliability for a correct OP-estimate then it fits well even if the OP-estimate is erroneous”.

Chapter 4

Verification vs. Reliability

Verification techniques selection requires a deep understanding of their features, their detection ability, of cost they entail, and of the features of the software being developed. When dealing with complex and wide systems, engineers have to cope with different and heterogeneous software faults. They are generally aware that a proper combination of verification techniques should be used, since applying just one technique is usually not sufficient to meet the dependability goal in an acceptable time. However, to date, it is not clear what are the most effective techniques to be applied to a system in hands and what techniques are more prone for detecting a specific kind of fault characterizing a system. This is left to the tester intuition and expertise. In this chapter we present an empirical comparison among verification techniques to provide a deeper knowledge of their characteristics in different situations. The study, in conjunction with what presented in the next chapter, intends to provide engineers with a support, based on quantitative evidences, for properly planning the verification techniques selection. Experimental campaigns described in the following two chapters provide encouraging results to pursue this direction.

4.1 Understanding the behaviour of Verification Techniques

The contribution presented in this chapter is an empirical analysis aiming to compare four widely-used verification techniques in critical systems development (i.e., functional testing, statistical testing, robustness testing, and stress testing), in order to establish their effectiveness in the detection of software faults and reliability improvement. The study intends to analyze, by running a controlled experimental campaign, the effectiveness of the mentioned techniques with respect to several factors, including the software type faults type by which the system is potentially affected. It considers the well-known Orthogonal Defect Classification (ODC), presented in chapter 1.1.3, as fault classification scheme, to distinguish fault types.

The Design of Experiments (DoE) methodology is used for experiments planning (16 experiments were run). Investigated aspects are the fault detection effectiveness, the effectiveness per fault type, the cost, and the delivered reliability. Results provide a deeper knowledge of the compared techniques in their ability to improve software quality under different initial conditions, giving valuable guidelines to engineers for techniques selection.

4.2 Design of an Empirical Analysis

4.2.1 Compared Techniques

Before deploying the final application, engineers need to test it against both functional and non-functional requirements, possibly using more than one technique. In this analysis, we compare four widely-used techniques for verification referred to as *Functional Testing, Statistical Testing, Robustness Testing and Stress Testing.* Even though all the mentioned techniques definitely aim to detect as many faults as possible, each of them pursues, in general, different goals and is able to improve some specific aspects of the software being developed. Here, we intend to compare their ability in detecting faults and improving reliability, and to investigate their differences with regard to the types of faults they are prone to detect.

In ***functional testing***, engineers derive test cases from specifications often using "the partition principle" (i.e., separating the input space in equivalence classes) along with boundary value analysis. It is often referred to as “specification-based testing” or “black-box testing” (as presented in Section 1.3.3), and it is described in many books and papers on software testing.

The goal of ***statistical testing*** is to improve the reliability of the software. Similarly to functional testing, statistical testing does not require any knowledge of the code; test cases are derived from an estimation of the operational profile, i.e., the set of inputs, and their frequency, that are expected in the operational phase. In order to provide a correct reliability prediction, a close estimate of the actual operational profile is required. Statistical testing is close to functional testing, since also in this case the input space is often partitioned in classes: the difference is that, in statistical testing, classes are assigned a probability of occurrence (from operational profile estimate) and test cases selection depends on this assignment.

Robustness testing evaluates the software application reaction to exceptional and unforeseen inputs. It provides a measure of how much the application is robust to unspecified inputs and how well it handles exceptional situations.

Stress testing evaluates the application’s ability to react to unexpected loads; even if the similarity with robustness testing is evident, stress testing does not use “exceptional inputs”; it uses normal input values, but with excessive load. In a sense, it lies in between robustness and functional testing.

The compared techniques can be grouped in two categories: both functional and statistical testing aim at demonstrating that the software does what it is required to do (we can call them ***"Demonstrative Testing"***), whereas robustness and stress testing aim at evaluating the application's ability to handle exceptional conditions by attempting to destruct the software (i.e., ***"Destructive testing"***). We want to evaluate how these differences affect the detection ability and reliability improvement, and how they manifest themselves in various experimental conditions.

4.2.2 Investigation Goals

The goal of this study is to characterize the testing techniques with respect to their: *(i) fault detection effectiveness, (ii) fault detection cost, (iii) detection effectiveness per fault type, and (iv) delivered reliability* (i.e., reliability that they make the software achieve). The output of this set of experiments will have to tell us how each technique behaves when applied to different software applications, characterized by different amounts and types of fault. In particular, the mentioned response variables are evaluated with respect to the following factors: *the (i) faults content that is initially present in the software, (ii) the percentages of fault types by which the software is affected, and (iii) the software type.* Experiments intend to answer the following questions:

Fault detection effectiveness

1. Which of the compared techniques detects the greatest percentage of faults (with respect to those initially present)?

2. Does the percentage of detected faults depend on the initial amount of faults in the program?

3. Does the percentage of detected faults depend on the software type?

Fault detection cost

1. Which of the techniques requires more testing time (measured in hours)? Which of them detects the greatest number of faults, standing the same testing time (i.e., the greatest #faults/hour ratio)?

2. Does the #faults/hour ratio (we call it detection rate) depend on the initial amount of faults?

3. Does detection rate depend on the software type?

The techniques are then compared with respect to their ability to detect faults of a given type. What we want to verify is the proneness of verification techniques in detecting faults of various type (among the considered ones). Hence, the questions we want to answer regard the effectiveness for each of the considered classes of faults are:

Fault detection effectiveness per fault type

1. Which of the compared techniques detects the greatest percentage of faults of a given type?

2. Is a given technique more prone to remove some type of faults than others?

3. Are faults of a given type more easily detectable with a particular technique than with another one?

Reliability

1. Which technique delivers the highest reliability?

2. How reliability achievement is affected by fault types initially present?

Experiments presented in the following sections allowed us to get useful (and statistically significant) answers for the outlined issues.

4.2.3 Experimental Procedure

The empirical study consists of the following phases:

i. Choosing the target applications. In order for experimental campaign to be reasonably representative, applications should not be *built-in-house* programs, but they should represent a wide class of applications. This would ease the generalization of results to other software applications. To meet this criterion we have chosen the following three widely adopted software applications: *(i) one of the most spread database management systems (DBMS), i.e.,* ***MySQL****; (ii) a commonly used Web server, i.e.,* ***Apache*** *and (iii) a networking application for Linux/Windows interoperability, i.e.,* ***Samba****.*

ii. Drawing the experimental plan. Once the target applications have been chosen, the experimental plan has been designed. In this phase, we have identified all the relevant factors and response variables needed to meet the investigation goals, and we have adopted the Design of Experiments (DoE) methodology, as detailed in the following subsection. The output of this step is a minimal list of treatments to be applied in order to get statistically significant answers.

iii. Injecting faults. In order to compare techniques with respect to fault types and faults content, the amount and types of faults present for each experiment need to be a "controllable factor", i.e., a value that we can determine. This implies a faults injection campaign to be performed, where a different amount of faults of each type is inserted into the software to be tested for each experiment, according to what the experimental plan suggests.

Thus the representativeness of injected faults, of their distribution in the software and of the injection technique are relevant issues. As for fault types, the ODC classification has been showed to represent common faults present in software applications. The distribution of faults adheres to the one actually observed in various studies about the presence of ODC fault types in several software programs [13], [12]; i.e., we used the same percentage as those actually observed in past studies, allowed to vary of a quantity of +/- 10%. This choice let us evaluate the impact of variations of different fault types on the response variables, considering realistic quantities.

The injection campaign has been repeated for each experiment, thus randomizing the place where faults are injected (i.e., minimizing the potential effect due to "where" faults are injected). The used injection technique is commonly known as "software mutation"(SM); it consists of applying changes at source code level, according to well-defined operators (for ODC operators see [13]).

iv. Running experiments. Each experiment tests the effectiveness of one technique applied to one software application with a given number of faults and a given combination of fault types. Hence, once the application under test is prepared with injected faults, the experiment is run.

v. Recording data. During each testing session the following data have been recorded: the number of detected faults (both absolute and per each type) and the percentages with respect to those initially present, the time when they were detected, and the test case number that exposed them.

vi. Measuring Reliability. To compare techniques with respect to the delivered reliability, we estimated the final reliability achieved for each testing session (i.e., for each experiment). The goal was to evaluate reliability that each technique produced in each experiment, and then analysing results. Details about reliability measurement are given in Section 4.4.

vii. Analyzing results. Collected data were then used in order to evaluate the variation of the output variables in response to the variations of the analyzed factors. Statistical tests allowed us to discard not meaningful results and to focus only on the most relevant relationships. After numerical interpretation of achieved results, the physical interpretation took place, by relating data to the questioned investigation goals.

4.2.4 Experimental Plan

The Design of Experiment (DoE) technique has been employed to plan the set of experiments. We adopted a fractional factorial design to keep the number of treatments low (hence, ignoring the analysis of interaction factors). Factors (i.e., the independent variables) considered in this study are: *i) testing technique, ii) faults content, iii) software type, and iv) the percentages of faults of a given type initially present in the application* (i.e., since we consider five faults type, there will be five additional factors). Factors are allowed to vary among the following levels:

- Testing technique can assume four distinct levels, i.e., *functional, statistical, stress, and robustness testing*;
- Software type can assume three levels: *MySQL, Apache, Samba*;
- Faults content can assume three levels: *low, medium and high.* The number of faults for the medium level for the three software applications has been determined by using a regression model, using software metrics as predictors. A regression model has been built by using eight software and twenty software metrics (the same ones presented in Chapter 5). The model presented an R^2 value of 0.73, meaning that the it explains the 73% of the variability of original data, and a *F-value* of 6.87 (i.e., its confidence level was greater than 95%, $p\text{-}value = 0.039 < \alpha = 0.05$). Extracting software metrics for MySQL, Apache and Samba and using the regression model, an estimate of faults content has been obtained. The low and high levels have been derived by varying the medium value of +/- 65 % (see Table 4.1).
- The percentages of fault types initially present are allowed to vary between a minimum and a maximum value (see Table 4.1). As mentioned, the percentage of faults of a given ODC type has been determined based on typical faults content of that type observed in real applications in past studies [13], [12]. In order to account for the prediction error, each quantity is allowed to vary between +/- 10% of the estimated value, that are the minimum and maximum values. These five variables are related by the constraint that their sum must amount to 100, since they are percentages values. Considering past empirical studies, we refer to realistic combinations of faults content,

instead of arbitrary, unlikely (and indefinitely numerous) combinations.

Factors and their levels are summarized in Table 4.1.

Table 4.1: Factors and levels adopted in the experiment. Legend: F = Functional Testing, ST = Statistical Testing, SS = Stress Testing, R = Robustness Testing

Initial faults content of type:		*Range*	
		From	To
Continuous Variables	Assignment	11%	31%
	Checking	15%	35%
	Interface	0%	17%
	Algorithm	30%	50%
	Function	0%	16%

		Levels			
Categorical Variables	**Technique**	*F*	*ST*	*SS*	*R*
	Faults #	*Low*	*Medium*		*High*
	Software	*MySQL*	*Apache*		*Samba*

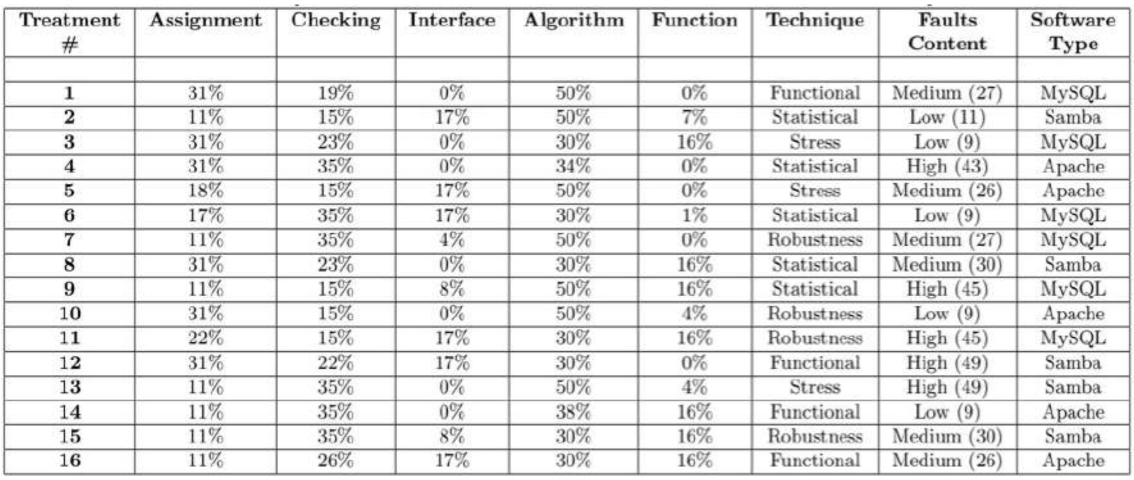
The design allows us to analyze the main effects of these factors on the response variables (i.e., the dependent variables. The experimental plan is reported in Figure 4.1.

The choice of factors and levels, and of response variables, enables the formulation of the following ANOVA model:

$$y = \mu + \sum_{i=1}^{8} \alpha_i + \epsilon \tag{4.1}$$

where:

- y is the response variable;
- μ is the mean response value;
- α_i variables are the main effects of the eight considered factors;



Treatment #	Assignment	Checking	Interface	Algorithm	Function	Technique	Faults Content	Software Type
1	31%	19%	0%	50%	0%	Functional	Medium (27)	MySQL
2	11%	15%	17%	50%	7%	Statistical	Low (11)	Samba
3	31%	23%	0%	30%	16%	Stress	Low (9)	MySQL
4	31%	35%	0%	34%	0%	Statistical	High (43)	Apache
5	18%	15%	17%	50%	0%	Stress	Medium (26)	Apache
6	17%	35%	17%	30%	1%	Statistical	Low (9)	MySQL
7	11%	35%	4%	50%	0%	Robustness	Medium (27)	MySQL
8	31%	23%	0%	30%	16%	Statistical	Medium (30)	Samba
9	11%	15%	8%	50%	16%	Statistical	High (45)	MySQL
10	31%	15%	0%	50%	4%	Robustness	Low (9)	Apache
11	22%	15%	17%	30%	16%	Robustness	High (45)	MySQL
12	31%	22%	17%	30%	0%	Functional	High (49)	Samba
13	11%	35%	0%	50%	4%	Stress	High (49)	Samba
14	11%	35%	0%	38%	16%	Functional	Low (9)	Apache
15	11%	35%	8%	30%	16%	Robustness	Medium (30)	Samba
16	11%	26%	17%	30%	16%	Functional	Medium (26)	Apache

Figure 4.1: The Experimental Plan

- ϵ is the experimental error random variable.

This model will allow to analyze how variations in each one of the factors affect the response variables, and hence to identify the most influential factors.

4.2.5 Fault Injection and Experiments Execution

As reported in Figure 4.1, 6 out of 16 experiments have been executed on MySQL, 5 out of 16 have been executed on Apache and the remaining 5 on Samba. Based on the percentages of fault of each type required by a treatment, a faults injection campaign has been carried out in order to prepare the software application for the test run, by using the mentioned technique (SM).

Then, the verification technique required by the plan has been applied. Each treatment was preceded by a preliminary phase to set up the test cases to be run: in some cases, this required a little work, either for the support of available tools or for the availability of part of test cases; in other cases we wrote test cases from scratch. In particular, both Samba and MySQL provide a valuable support for executing tests: they allow to execute existing test suites with easy-to-use scripts, to modify them and to write new test cases quite easily; Apache instead required more effort, since its support to testing is growing, but still limited. As for MySQL, we used some scripts provided with the distribution as well as a tool, namely *MySQLslap*, for test cases execution. In this case, robustness and stress tests were written from scratch, whereas functional test cases definition required some modifications to the existing test suites. Statistical testing required the definition of an operational profile (as with Apache and Samba too).

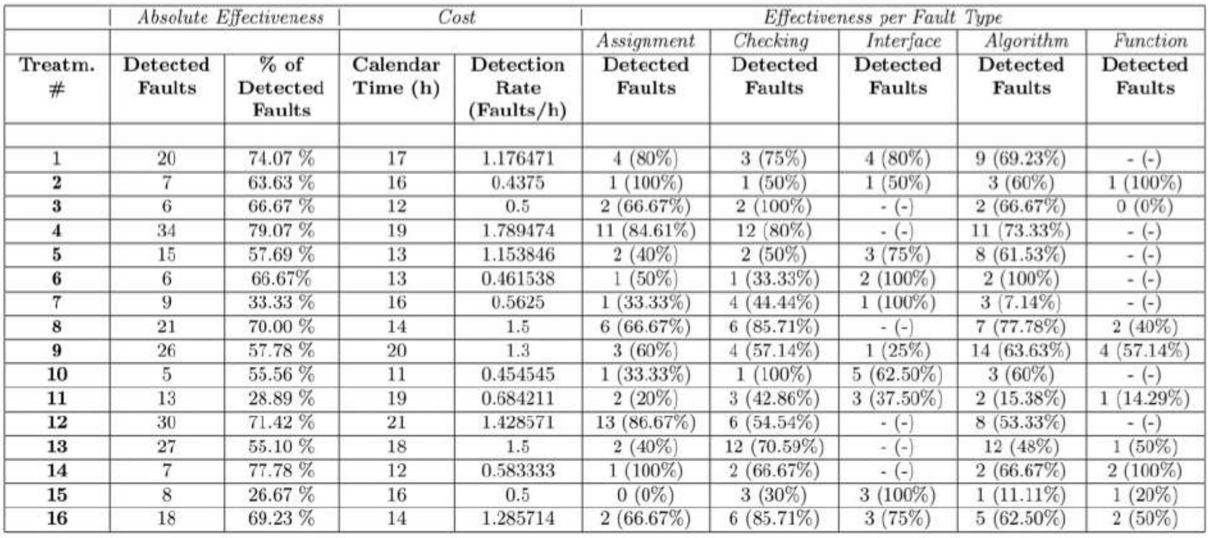
As for Samba, we used smbTorture utility and supporting scripts, provided with the distribution, to test the software with all the four techniques. With robustness and stress testing, we extended the existing test suites because it was insufficient compared with test suites in the other two applications.

Finally, as for Apache, the tool *JMeter* was used, in order to write and execute test cases. The test cases for all of the four techniques were written from scratch in this case (*JMeter* made these operations easier).

4.3 Data Analysis - Fault Detection Perspective

In this section, data collected during experiments are analyzed with respect to the first three investigation goals. In particular, response variables regarding fault detection effectiveness, cost and detection effectiveness per type are discussed; results on reliability are discussed in the next section.

Figure 4.2 reports a summary of obtained results. The first column reports the treatment number (the combination of factor levels for each treatment can be seen in Figure 4,1, which reports the experimental plan). The second and third column report, respectively, the total number and the percentage of faults detected in the corresponding experiment (i.e., the response variables to measure the Absolute Effectiveness); the third and fourth column report, respectively, the calendar time, measured in hours, and the detection rate, measured in number of faults per hour (i.e., the response variables referring to the Cost). The remaining five columns report the number (and the percentage) of detected faults per each type. Some of these columns have no value in some cells, meaning that, in the corresponding treatment, the value for that factor had to set to 0 (e.g., in the first treatment, the number of function faults to inject is 0; thus no function faults could be detected and the corresponding cell in Figure 4.2 reports no value).



	Absolute Effectiveness		*Cost*		*Effectiveness per Fault Type*				
					Assignment	*Checking*	*Interface*	*Algorithm*	*Function*
Treatm. #	**Detected Faults**	**% of Detected Faults**	**Calendar Time (h)**	**Detection Rate (Faults/h)**	**Detected Faults**	**Detected Faults**	**Detected Faults**	**Detected Faults**	**Detected Faults**
1	20	74.07 %	17	1.176471	4 (80%)	3 (75%)	4 (80%)	9 (69.23%)	- (-)
2	7	63.63 %	16	0.4375	1 (100%)	1 (50%)	1 (50%)	3 (60%)	1 (100%)
3	6	66.67 %	12	0.5	2 (66.67%)	2 (100%)	- (-)	2 (66.67%)	0 (0%)
4	34	79.07 %	19	1.789474	11 (84.61%)	12 (80%)	- (-)	11 (73.33%)	- (-)
5	15	57.69 %	13	1.153846	2 (40%)	2 (50%)	3 (75%)	8 (61.53%)	- (-)
6	6	66.67%	13	0.461538	1 (50%)	1 (33.33%)	2 (100%)	2 (100%)	- (-)
7	9	33.33 %	16	0.5625	1 (33.33%)	4 (44.44%)	1 (100%)	3 (7.14%)	- (-)
8	21	70.00 %	14	1.5	6 (66.67%)	6 (85.71%)	- (-)	7 (77.78%)	2 (40%)
9	26	57.78 %	20	1.3	3 (60%)	4 (57.14%)	1 (25%)	14 (63.63%)	4 (57.14%)
10	5	55.56 %	11	0.454545	1 (33.33%)	1 (100%)	5 (62.50%)	3 (60%)	- (-)
11	13	28.89 %	19	0.684211	2 (20%)	3 (42.86%)	3 (37.50%)	2 (15.38%)	1 (14.29%)
12	30	71.42 %	21	1.428571	13 (86.67%)	6 (54.54%)	- (-)	8 (53.33%)	- (-)
13	27	55.10 %	18	1.5	2 (40%)	12 (70.59%)	- (-)	12 (48%)	1 (50%)
14	7	77.78 %	12	0.583333	1 (100%)	2 (66.67%)	- (-)	2 (66.67%)	2 (100%)
15	8	26.67 %	16	0.5	0 (0%)	3 (30%)	3 (100%)	1 (11.11%)	1 (20%)
16	18	69.23 %	14	1.285714	2 (66.67%)	6 (85.71%)	3 (75%)	5 (62.50%)	2 (50%)

Figure 4.2: Experimental Results

Table 4.2: Results of detection effectiveness, grouped by factors

Factors	**Average % of Faults**
Grouped by Testing Technique	
Functional	73.13%
Statistical	67.43%
Stress	59.82%
Robustness	36.11%
Grouped by Software Type	
MySQL	54.47%
Apache	67.87%
Samba	57.36%
Grouped by Faults Content	
Low	66.06%
Medium	55.17%
High	58.45%

4.3.1 Detection Effectiveness

Since the variable "number of detected faults" is affected by the number of faults that are initially injected, we evaluate the effectiveness of faults detection as the percentage of detected faults with respect to those initially injected. Table 4.2 reports the data aggregated by the three factors of testing techniques, software type and initial faults content.

Observing data aggregated by testing techniques, it is worth noting that, in the average, *(i) functional and statistical testing detect a greater percentages of faults than stress and robustness testing; (ii) stress testing seems to be closer, from this point of view, to those techniques that we called "Demonstrative testing" than to robustness testing. (iii) It is evident that robustness testing detects, in the average, fewer faults than the others; one explanation is that robustness testing goal is to detect the "hardest", uncommon faults, even* *if few, rather than many common faults.*

In particular, hypothesis tests report that:

- *The difference between functional and statistical testing (i.e., 5.69%) is not statistically significant (i.e., the value "functional - statistical" was not statistically different form 0 at a significance level of* $\alpha > 0.05$*, i.e., p-value = 0.11);*
- *Functional testing detected 13.30% more faults than stress testing (at $\alpha <$ 0.01), and 37.01% more faults than robustness testing (at $\alpha <$ 0.01);*
- *The difference between statistical testing and stress testing (7.61%) is not statistically significant ($\alpha >$ 0.05, p-value = 0.10); whereas the difference between statistical testing and robustness testing (31.32%) is significant at $\alpha <$ 0.01;*
- *Stress testing detected 23.71% more faults than robustness testing (at $\alpha <$ 0.05).*

Hence, functional testing is confirmed to behave better than both stress and robustness testing; statistical testing overcomes only robustness testing, as also stress testing. Functional and statistical testing have similar performances. The difference, even if not significant, can be explained by considering that statistical testing aims to improve reliability, exercising mainly the part of code more likely exercised at runtime; functional testing covers a wider portion of the code (a similar explanation may stand for stress testing, since it exercises some common functionalities with overloaded inputs, but does not span over the code).

In the second part of Table 4.2, data are aggregated by software type; by adopting further hypothesis tests, we gain insights about the dependence of detected faults on software type. In particular, we can observe that the differences in faults percentage detected in Apache and Samba (i.e., 10.51 %), Apache and MySQL (i.e., 13.4%), and Samba and MySQL (i.e., 2.89 %) are not statistically significant ($\alpha > 0.05$ for all the tests, with p-value = 0.15, p-value = 0.10 and p-value = 0.40, respectively). This means that software type did not affect the ability of testing techniques to detect faults.

The representativeness of the chosen applications in their respective classes (i.e., DBMSs, Web Servers and Network protocols) along with these results suggest that observed differences among techniques in detecting faults may, to some extent, be generalized (at least inside the considered classes of software).

Similarly, the last part of Table 4.2 reports the same data aggregated by the initial faults content factor. We can derive that the difference in faults percentage detected with an initial low faults content and an initial high faults content (7.61%) is not statistically significant ($\alpha > 0.05$, p-value = 0.22). Hence, the initial faults content has no significant impact on the percentage of detected faults.

Thus, from this analysis, the only factor affecting the detection effectiveness is the testing technique.

Note that reported hypothesis tests comparing means are t-student tests; all the results have been also confirmed by performing the Tukey's test at 95% protection level, as also results in successive sections.

The impact of testing techniques on detection effectiveness, as well as the non-influence of

Table 4.3: Results of cost response variables, grouped by factors

Factors	**Average Calendar Time (h)**	**Average Detection Rate (Faults/h)**
Grouped by Testing Technique		
Functional	16	1.118522409
Statistical	16.4	1.097702429
Stress	14.33	1.051282051
Robustness	15.5	0.550313995
Grouped by Software Type		
MySQL	16.16	0.780786596
Apache	13.8	1.053382582
Samba	17	1.073214268
Grouped by Faults Content		
Low	12.8	0.48738345
Medium	15	1.029755171
High	19.4	1.340451128

other factors, has been confirmed by the analysis of variance (ANOVA), carried out by the well-known tool JMP. These further statistical confirmations are omitted for brevity.

4.3.2 Cost

Variables measuring fault detection cost are calendar time (in hours) and fault detection rate, measured in terms of number faults detected per hour. Table 4.3 reports data aggregated by the three factors of testing technique, software type and initial faults content.

Results of hypothesis tests show that:

- The differences in calendar testing time between the technique that required, in the average, the most time (i.e., Statistical testing) and the technique that required the least time (i.e., Stress testing) is 2.067 hours. It is not statistically significant (α >0.05, p- value = 0.19); in the average, there are no significant differences in required calendar time;

- On the other hand, the difference between the technique with the highest detection rate (that has been the functional testing) and the technique with the lowest one (i.e., the robustness testing), which is of 0.5682 Faults/hour, is statistically significant ($\alpha <$ 0.05). One possible explanation is that robustness testing, other than detecting fewer faults than the other techniques, requires testing to be stopped more often (because even when the application correctly handles the given exceptional inputs, it often stops and exits, correctly reporting the occurred exception; thus testing has to be restarted often).

In the second part of Table 4.3, data are aggregated by software type; we can investigate about the dependence of detected faults on software type:

- *The difference between calendar time values in Samba and Apache (i.e., 3.2h) is not statistically significant ($\alpha >$ 0.05, p-value = 0.059);*
- *The difference between calendar time values in MySQL and Apache (i.e., 2.36h) is not statistically significant ($\alpha >$ 0.05, p-value = 0.12);*
- *The difference between calendar time values in Samba and MySQL (i.e., 0.83h) is not statistically significant ($\alpha >$ 0.05, p-value = 0.32);*
- *The difference between the highest mean detection rate (observed in Samba) and the lowest one (observed in MySQL), that is of 0.2924 faults/hour, is not statistically*

significant ($\alpha > 0.05$, p-value = 0.15);

Hence, also in this case there is no difference due to the software type in variables measuring fault detection cost.

Finally, the last part of Table 4.3 reports the same data aggregated by the initial faults content factor. We can observe that:

- Applications with highest initial faults content require more time than those with low initial faults content (the average difference has been 6.6h, significant at $\alpha < 0.01$); this is quite intuitive.
- Applications with highest initial faults content experienced, in the average, a higher detection rate than those with low initial faults content (the average difference has been 0.8530 faults/hour, significant at $\alpha < 0.01$). One possible explanation is that at the beginning of the test, in software with high initial faults content more faults are exposed and the initial rate is very high; then, as faults are removed, the behaviour in the two cases should be almost the same.

4.3.3 Effectiveness per Type

The last point is to investigate the effectiveness in detecting faults of different types. The same response variable has been used as in the first point: i.e., the percentage of faults detected per type.

Table 4.4 reports data aggregated by the three factors of testing technique, software type, and initial faults content for each of the considered fault categories.

Table 4.4: Results of detection effectiveness per fault type, grouped by factors

Factors	**% of Detected Faults**				
	Assignment	**Checking**	**Interface**	**Algorithm**	**Function**
Grouped by Testing Technique					
Functional	83.34%	70.48%	64.17%	62.93%	75.00%
Statistical	60.21%	61.24%	58.33%	74.95%	65.71%
Stress	48.89%	73.53%	75.00%	58.73%	25.00%
Robustness	21.67%	54.33%	87.50%	23.41%	17.15%
Grouped by Software Type					
MySQL	51.67%	58.80%	73.50%	53.68%	23.81%
Apache	64.92%	76.48%	75.00%	64.81%	75.00%
Samba	58.67%	58.17%	62.50%	50.04%	52.50%
Grouped by Faults Content					
Low	70.00%	70.00%	75.00%	70.67%	66.67%
Medium	47.78%	61.81%	86.00%	48.22%	36.67%
High	58.26%	61.03%	41.67%	63.42%	40.48%

Results show that, in the average, functional testing is the fairest technique (it detects comparable percentages of distinct fault types), and that robustness and stress testing are more prone to remove some kinds of fault rather than others. In particular, we observed that:

- Functional testing detects percentages of faults for each type that do not differ significantly among each other; (as may be seen in the first row of Table 4.4); the only exception is the percentage of Assignment faults, that is significantly higher than the others (e.g., the difference between the percentage of Assignment faults and Algorithm faults (i.e., 20.41%) is significant at $\alpha < 0.05$). This means that functional testing is quite fair with respect to the different fault types;
- Robustness testing detects 65.83% more faults of Interface type than Assignment, 64.09% more faults than Algorithm, 70.35% more faults than Function type, and

33.17% more faults than Checking type ($\alpha < 0.01$ for the difference with Assignment faults, $\alpha < 0.05$ for the others); also the percentage of Checking type is high with respect to the others, but at a significance level lower than 95%;

- Some other differences are worth to mention, even though the hypothesis tests do not provide significance level greater than 95% due to the lack of data for some treatments (for instance, Function faults are absent in several treatments, due to the their relative low occurrence in real applications): in particular, it is possible to see that in the average, robustness testing detects more interface faults than functional testing and statistical testing too (24% and 30%, respectively). Stress testing also detects many faults of Interface type (75%) and of Checking type (73.53%) with respect to the other types. Functional testing (and statistical testing too) detects many more Function faults than robustness testing (57.85% and 48.56%, respectively).

4.4 Data Analysis - Reliability Perspective

We already mentioned that detecting more faults does not necessarily imply improving reliability. Thus, to compare techniques from reliability perspective, further measurements have been performed. In particular, for each of the 16 experiments (showed in Figure 4.1), we assessed the reliability achieved at the end of that testing session, i.e., after that a given technique has been applied.

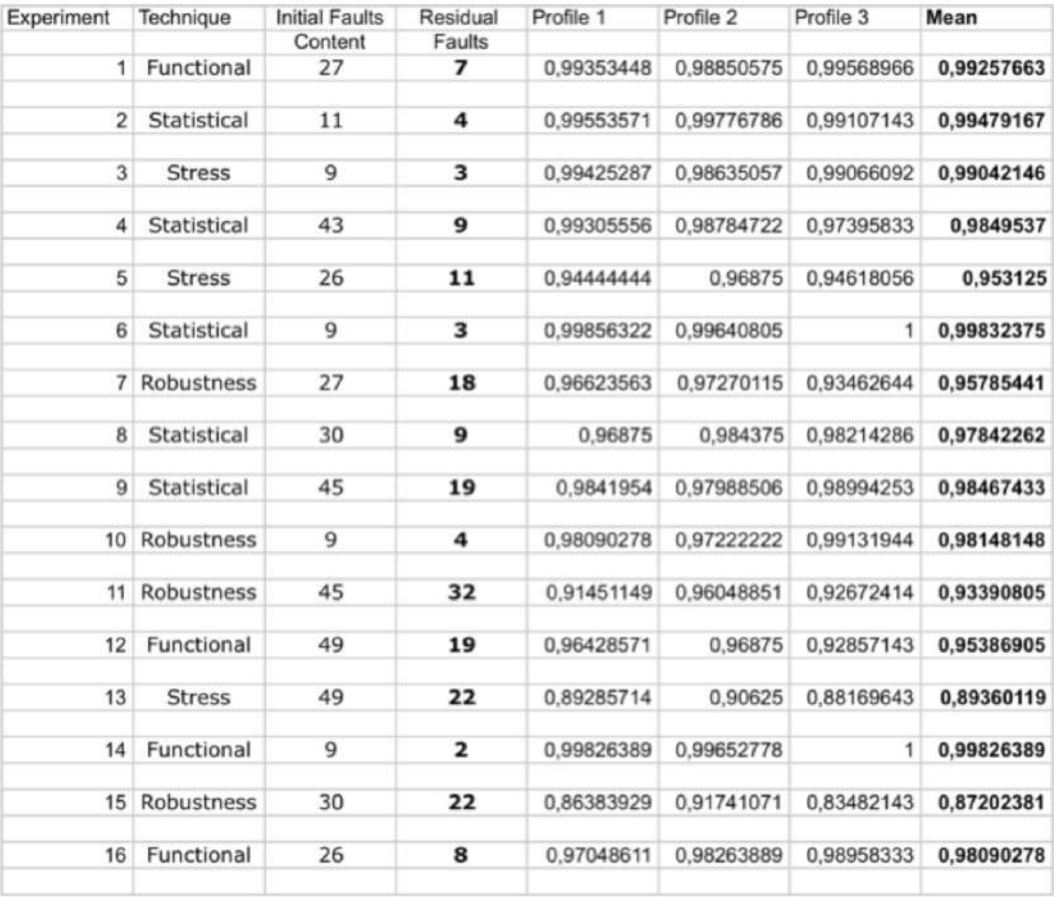
Since reliability is sensitive to the operational profile, we generated for each software application (i.e., for MySQL, Samba, Apache) three distinct operational profiles. The set of operations that each application could perform was split in equivalence classes; assuming a uniform distribution inside each class, we then assigned distinct occurrence probability values to each of them. A measurement consists of an execution where operations are randomly picked up from the classes according to the defined operational profile distribution. Picking inputs from three different profiles, the final estimate averages the effect due to three possible distinct usage of the application.

At the end of the execution, reliability was measured with the frequency interpretation approach, i.e., according to the expression (2) in chapter 1, and as in [44]: $R \approx 1 - \lim_N Nf/N$, where Nf is the number of experienced failures and N is the number of randomly generated operations (i.e., the inputs). To use this method correctly, when a failure occurs during the execution, the corresponding fault has not to be removed (i.e, the system has not to be altered during the measurement).

Figure 4.3 shows reliabilities achieved after each treatment and for each operational profile. In the last column the average reliability obtained in each case is reported.

Rows represent treatments where the corresponding technique, reported in the third column, has been applied. The application of the technique removed a given amount of faults, causing the initial faults content (second column) to be reduced. Reliability measurement is then applied to the software application version with the residual content of faults reported in the third column.

Of course, experiments where a technique left few faults show, in the average, a high reliability; however the differences in the measurements taken from different profiles shows that even with few faults, reliability can be low (as in the experiment number 10, with the second profile). Similarly, the same software application may exhibit a quite high reliability



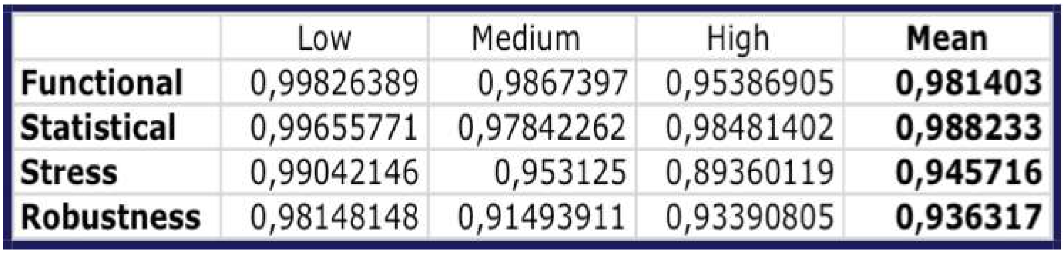
Experiment	Technique	Initial Faults	Residual	Profile 1	Profile 2	Profile 3	**Mean**
		Content	Faults				
1	Functional	27	**7**	0,99353448	0,98850575	0,99568966	**0,99257663**
2	Statistical	11	**4**	0,99553571	0,99776786	0,99107143	**0,99479167**
3	Stress	9	**3**	0,99425287	0,98635057	0,99066092	**0,99042146**
4	Statistical	43	**9**	0,99305556	0,98784722	0,97395833	**0,9849537**
5	Stress	26	**11**	0,94444444	0,96875	0,94618056	**0,953125**
6	Statistical	9	**3**	0,99856322	0,99640805	1	**0,99832375**
7	Robustness	27	**18**	0,96623563	0,97270115	0,93462644	**0,95785441**
8	Statistical	30	**9**	0,96875	0,984375	0,98214286	**0,97842262**
9	Statistical	45	**19**	0,9841954	0,97988506	0,98994253	**0,98467433**
10	Robustness	9	**4**	0,98090278	0,97222222	0,99131944	**0,98148148**
11	Robustness	45	**32**	0,91451149	0,96048851	0,92672414	**0,93390805**
12	Functional	49	**19**	0,96428571	0,96875	0,92857143	**0,95386905**
13	Stress	49	**22**	0,89285714	0,90625	0,88169643	**0,89360119**
14	Functional	9	**2**	0,99826389	0,99652778	1	**0,99826389**
15	Robustness	30	**22**	0,86383929	0,91741071	0,83482143	**0,87202381**
16	Functional	26	**8**	0,97048611	0,98263889	0,98958333	**0,98090278**

Figure 4.3: Results of reliability measurements

even having a considerable amount faults: for instance, experiment number 9 on MySQL, stressed according to the third profile show a greater reliability than experiment number 1 (profile 2) on the same application, even though in the first case a greater amount of faults was left by the testing technique (19 in the former case, 7 in the latter). "Fewer faults" does not guarantee higher reliability, even if in the average this is true. Hence, fault detection and reliability perspective often show the same trend, but these results confirm that it is not always the case. Choosing techniques to improve reliability may be different than choosing techniques to reduce the number of faults.

As for techniques comparison, Figure 4.4 reports the average reliability obtained by each technique. Obtained reliabilities show that:

- *Functional testing leads to a mean reliability greater than stress testing (0.0356, i.e., 3.77%) at a significance level greater than 95% (p-value = 0,0201)*
- *Statistical testing is superior to stress testing, bringing a reliability improvement of 0.0425 (i.e., 4.49 %) at a significance level greater than 99% (p-value =0,0037)*
- *Statistical testing overcomes robustness testing of 0.0519 (i.e., 5.54 %) at significance level greater than 99% (p-value = 0,0023)*
- *Also functional testing performs better than robustness testing from reliability perspective (the difference is 0.045, i.e., 4.81 %, with p-value = 0,0103)*
- *Statistical testing is slightly better than functional testing but the difference is not significant (the difference is 0.006, i.e., 0.69 %, with p-value =0,144)*



	Low	Medium	High	**Mean**
Functional	0,99826389	0,9867397	0,95386905	**0,981403**
Statistical	0,99655771	0,97842262	0,98481402	**0,988233**
Stress	0,99042146	0,953125	0,89360119	**0,945716**
Robustness	0,98148148	0,91493911	0,93390805	**0,936317**

Figure 4.4: Measurements of Average Reliability

The main difference with fault detection effectiveness is about the last test: statistical testing performed slightly better than functional testing from reliability perspective, even if not significantly.

Observing the results, it is also evident that the average reliability depends on the amount of faults initially present. Figure 4.4 reports the obtained reliability under different initial conditions. Almost always, a greater amount of initial faults led to a lower final reliability. But this is not always the case: for instance, statistical testing performed better with an initially high faults content than with a medium faults content.

4.5 Discussion

Results confirmed our hypothesis, which is also a common sense, that verification techniques "systematically" behave differently when the mentioned factors vary. In particular, the analysis highlighted that the effectiveness is dependent on faults content and fault types variation (in distinct ways for the various techniques), but less on the variation of software applicative target.

In summary, results showed that Functional testing is able to detect a greater percentage of faults than stress and robustness testing, and it seems to be the fairest technique with respect to the fault types it is prone to detect; its difference with statistical testing is not significantly relevant. Statistical testing also detects more faults than robustness testing (but it does not significantly overcome stress testing) and it also turned out to be a 'fair" technique. It differs from functional testing essentially for the different goal that it pursues: aiming at improving reliability, statistical testing stresses with higher probability a given part of code, i.e., the one more likely exercised at runtime, considering less the unlikely functionalities. This is confirmed by the results regarding reliability, which showed a substantially comparable behaviour between these techniques, but with statistical testing that in this case, slightly over-performed functional testing.

Functional and statistical testing also highlighted a better ability in detecting faults of Function type, in contrast to stress and robustness testing that are not able to deal with such kinds of fault.

Stress testing detects a good percentage of faults; it also focuses on common functionalities, not considering all the code. Although this may penalize it in the absolute number of detected faults, it allows to better detecting some specific kind of faults. In particular, stress testing exhibits good performances in detecting Checking faults and Interface faults. From reliability's point of view, it behaves similarly to the detection ability perspective, i.e., it provided higher reliability than robustness testing, but substantially worse than the functional and statistical testing.

Robustness testing, even detecting fewer faults and delivering lower reliability than the others, is turned out as the best one to remove faults of Interface type (for instance, it is very good at detecting faults on passed parameters) and of Checking type. This makes it an important technique, since it complements the others in the detection of faults that are hard to be exposed, e.g., by functional or statistical testing.

It is worth to remark that the obtained results refer to the fault detection and reliability improvement brought by techniques when these are applied alone. Hence, it is wrong to derive from results that robustness testing must not be applied to improve reliability; results say that robustness testing should not be applied alone.

From reliability's point of view this is a very important issue. Indeed, verification techniques ultimately determine what types of fault remain in the software, which in turn affect reliability.

For instance, robustness testing will remove more likely the interface and checking faults, as showed above. The final reliability depends on the occurrence probability of residual faults. Indeed, faults activated under exceptional conditions (as those removed from robustness or stress testing), have low occurrence probability; hence such techniques if applied alone cannot improve reliability, since they do not aim to remove high-occurrence faults.

On the other hand, applying functional or statistical testing alone, improves reliability just up to a given bound: when they removed high-occurrence faults, such techniques stabilize, and reliability is no longer improvable in reasonable testing time (since they are less prone to remove the residual low occurrence faults). This becomes a tough problem for critical systems, where high reliability levels are required.

To boost reliability beyond this limit, these techniques should therefore be suitably combined to span over the entire code and fault types. How to select them is the topic discussed in chapter 6.

Chapter 5

Software System Characterization

The knowledge of verification techniques with respect to their detection ability, their delivered reliability and their behaviour against different fault types is of paramount importance for engineers responsible for setting a verification strategy. The analysis presented in chapter 4 was conceived to provide such knowledge. However, verification strategy does not depend only on the features of techniques that could be selected; it also depends on how well such techniques are suitable for the specific software being developed. This requires, jointly with what presented in chapter 4, a deep knowledge of the software under test. This chapter presents our approach to characterize the system by the tester's point of view. We first present an empirical analysis carried out to evaluate potential relationships between some relevant software's features and the ODC fault types affecting a software system. The features of the software were expressed by means of common software metrics. Such a characterization adopted techniques of fault-proneness models to predict faults content from software metrics. Then, by adopting the same principle and statistical techniques, we considered an extended set of features, by additional metrics, to also characterize the system with respect to the phenomenon of software aging. A further empirical analysis to relate software's features to software aging is presented; this allows engineers to have a preliminary prediction of software aging proneness in the testing phase. Results of these analyses, jointly with what presented in chapter 4, can be very useful to testing practitioners who can start by characterizing the system by means of software metrics; then they can obtain a prediction of fault types that potentially affect the system and of aging proneness of modules; and, finally, they can choose the most effective combination of verification techniques (and focus greatest "aging detection" effort) to provide a better detection coverage of software faults.

5.1 Software Metrics vs. Fault Types

5.1.1 Investigation goals

The analysis presented in chapter 4 showed the existence of some relevant relationships between verification techniques and the amount and types of faults potentially present in the software. Such relationships could be important for a tester to select one technique (or a convenient combination of techniques), since s/he could adopt the best strategy based on fault content and types expected to be present in the software under test.

However, tester cannot easily know what kinds of fault characterize the software; this depends on how the software has been developed, on what programming techniques have been adopted, on how complex and large the developed code is and on other factors. Since many of software features can be expressed by common software metrics, we investigated the potential relationship between fault types present in the software (classified according to the ODC) and a set of common software metrics. The conjecture here is that metrics are able to capture (and to describe) those software features that mainly determine (or are related to) what types of faults affect a software application.

If such a relation exists, tester is able to give a statistical estimate of fault content and types that characterize the software; on that base s/he adopts the most suitable techniques for the verification phase, by exploiting the results previously presented. As discussed in Section 2.1, several studies in the literature ascertained a statistical relationship between software metrics and faults content in the software; here we aim to extend such studies, by inferring potential relationships between metrics and ODC fault types.

Thus, the main questions to be answered by this study are:

- ***Does it exist some relation between software metrics and ODC fault types?***
- ***If yes, is it possible to give estimates of ODC faults content in the software by using its metrics?***

5.1.2 Empirical Analysis

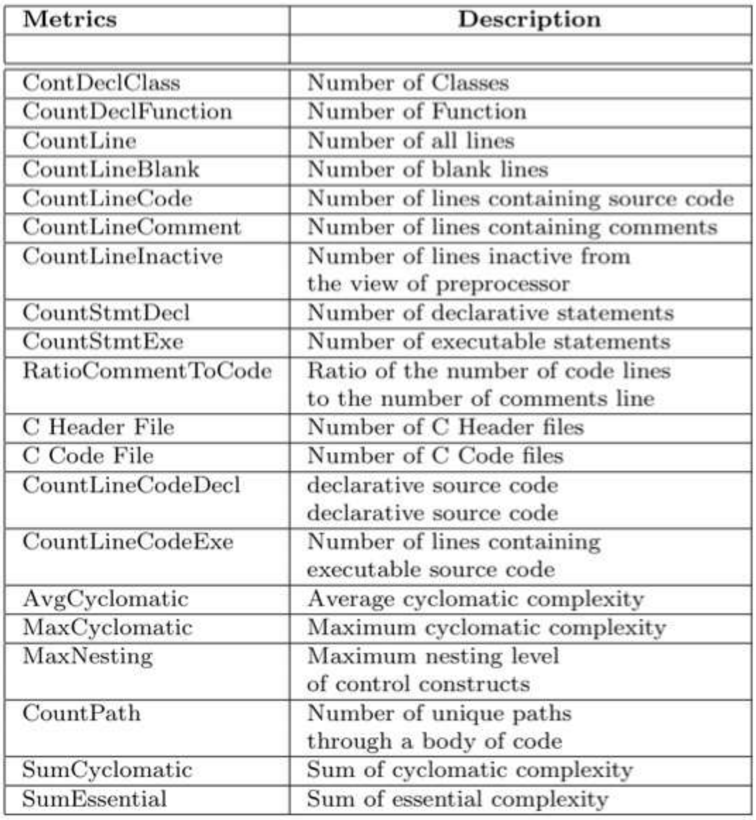
The empirical analysis has been carried out following these steps:

- *Metrics Selection*, where a set of metrics is chosen as predictor variables;
- *Software Selection*, where a set of software applications with known values of the response variables are selected as samples;
- *Metrics Extraction*, where static analysis tools are adopted to extract metric values of selected software;
- *Verifying correlation*, where a first rough analysis is performed to evaluate whether metrics and ODC fault types are correlated;
- *Building of regression models*, where models to predict response variables from independent variables are formulated.

Metrics Selection

A set of 20 common software metrics describing the code complexity and size are selected as predictors. Figure 5.1 shows the chosen metrics with a brief description.

These metrics can be roughly categorized as metrics describing the code volume (that are the majority), such as *CountLine, CountLineBlank, CountLineCode, CountLineComment, CountLineCode Decl, CountLineCodeExe*, metrics describing the files volume, such as *C Header Files, C Code Files,* and metrics describing complexity, such as *AvgCyclomatic, MaxCyclomatic, MaxNesting, CountPath, SumCyclomatic, SumEssential.* Metrics



Metrics	Description
ContDeclClass	Number of Classes
CountDeclFunction	Number of Function
CountLine	Number of all lines
CountLineBlank	Number of blank lines
CountLineCode	Number of lines containing source code
CountLineComment	Number of lines containing comments
CountLineInactive	Number of lines inactive from the view of preprocessor
CountStmtDecl	Number of declarative statements
CountStmtExe	Number of executable statements
RatioCommentToCode	Ratio of the number of code lines to the number of comments line
C Header File	Number of C Header files
C Code File	Number of C Code files
CountLineCodeDecl	declarative source code declarative source code
CountLineCodeExe	Number of lines containing executable source code
AvgCyclomatic	Average cyclomatic complexity
MaxCyclomatic	Maximum cyclomatic complexity
MaxNesting	Maximum nesting level of control constructs
CountPath	Number of unique paths through a body of code
SumCyclomatic	Sum of cyclomatic complexity
SumEssential	Sum of essential complexity

Figure 5.1: Selected metrics

as *CountDeclClass and CountDeclFunction*, could be mentioned in the code volume metrics; however, they describe something more than the "size" of the code. A high number of classes and/or function may indicate a large code volume, but may also indicate high modularity (not necessarily coupled with a large code volume).

Software Selection and Metrics Extraction

To evaluate relations between metrics and ODC fault types, a set of software applications of which we can know the metrics value and the ODC fault types content is needed as sample

Table 5.1: Faults Content per Type in the chosen Software Applications

Software	**Fault Types**				
	Assign.	*Checking*	*Interface*	*Algor.*	*Function*
Linux	21	24	5	31	12
Bash	2	0	0	0	0
Pdftohtml	11	1	0	8	0
Firebird	1	1	0	0	0
Scummvm	18	6	3	42	5
Zsnes	2	0	0	1	0
Freecvi	6	7	4	28	8
Mingwrt	6	23	3	28	0

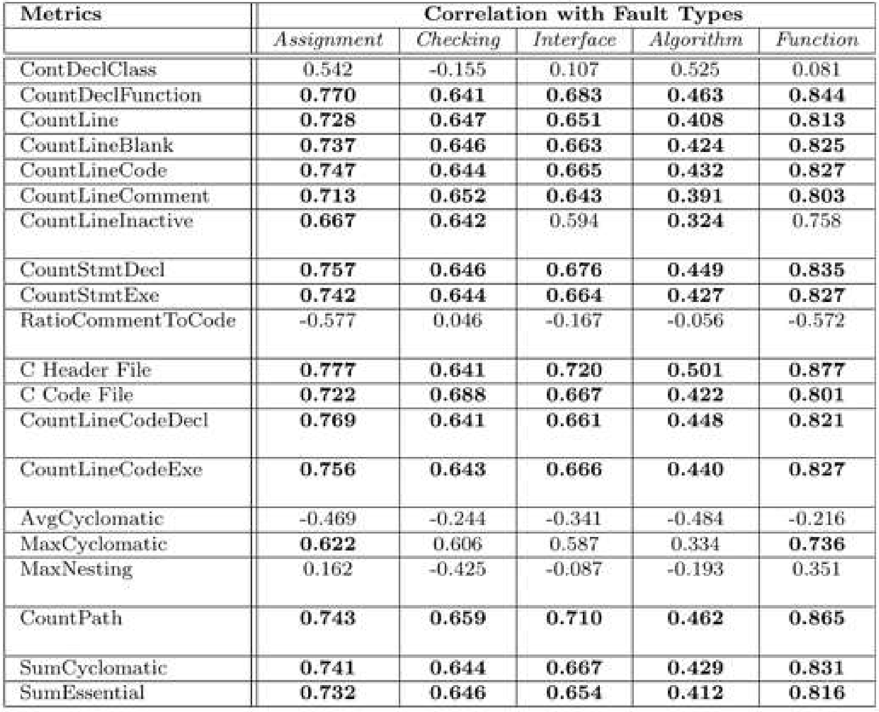
to build regression models. We have chosen a set of eight software applications with known ODC fault types content: they were the subjects of the study in [13], where a classification task from bug reports was carried out by authors for *fault injection* -related purposes. From the initial set of that study (twelve applications), we discarded three applications as possible outliers and one application since no longer available, obtaining a final set of eight software applications, shown in Table 5.1.

Columns report the number of faults per type affecting each of the chosen software. Once chosen the software, metric values need to be extracted. We did it by using a tool for static code analysis (named Understand), which derived the values for the 20 metrics selected in the first step.

Regression Models

When both data about metrics and faults content were available, we have verified, before building regression models, if software metrics correlate with ODC fault types content.

Results are reported in Figure 5.2, confirming the presence of statistically significant correlations for several metrics. Cells highlighted in boldface indicate the presence of correlation at 0.05 significance level. This allows us to answer the first question: ***there is a correlation between software metrics and ODC fault types.***



Metrics	**Correlation with Fault Types**				
	Assignment	*Checking*	*Interface*	*Algorithm*	*Function*
ContDeclClass	0.542	-0.155	0.107	0.525	0.081
CountDeclFunction	**0.770**	**0.641**	**0.683**	**0.463**	**0.844**
CountLine	**0.728**	**0.647**	**0.651**	**0.408**	**0.813**
CountLineBlank	**0.737**	**0.646**	**0.663**	**0.424**	**0.825**
CountLineCode	**0.747**	**0.644**	**0.665**	**0.432**	**0.827**
CountLineComment	**0.713**	**0.652**	**0.643**	**0.391**	**0.803**
CountLineInactive	**0.667**	**0.642**	0.594	**0.324**	0.758
CountStmtDecl	**0.757**	**0.646**	**0.676**	**0.449**	**0.835**
CountStmtExe	**0.742**	**0.644**	**0.664**	**0.427**	**0.827**
RatioCommentToCode	-0.577	0.046	-0.167	-0.056	-0.572
C Header File	**0.777**	**0.641**	**0.720**	**0.501**	**0.877**
C Code File	**0.722**	**0.688**	**0.667**	**0.422**	**0.801**
CountLineCodeDecl	**0.769**	**0.641**	**0.661**	**0.448**	**0.821**
CountLineCodeExe	**0.756**	**0.643**	**0.666**	**0.440**	**0.827**
AvgCyclomatic	-0.469	-0.244	-0.341	-0.484	-0.216
MaxCyclomatic	**0.622**	0.606	0.587	0.334	**0.736**
MaxNesting	0.162	-0.425	-0.087	-0.193	0.351
CountPath	**0.743**	**0.659**	**0.710**	**0.462**	**0.865**
SumCyclomatic	**0.741**	**0.644**	**0.667**	**0.429**	**0.831**
SumEssential	**0.732**	**0.646**	**0.654**	**0.412**	**0.816**

Figure 5.2: Selected metrics

Standing correlation results, the next step is the definition of statistical regression models able to predict the amounts of faults of each ODC type, by using software metrics as predictors. One difficulty associated with the models construction is determined by the high inter-correlation among some metrics (and hence among the predictors): this causes the well-known problems of multicollineraity. For instance, *CountLine* and *CountLineBlank* are strongly correlated among each other; this leads to an inflated variance in the estimation of the dependent variables.

To overcome this problem, two techniques are usually adopted: the *Principal Component Analysis* (PCA) or the *Stepwise regression.* The first one transforms original data into uncorrelated data; it computes new variables, called Principal Components (PCs), which are linear combination of original variables, such that all principal components are uncorrelated. From this new set of variables, a subset of them able to explain the most of variance of original data is usually selected. Typically, a very small percentage of original variables (e.g.: 10%) are able to explain 85% to 90% of the original variance. Each of the principal components is expressed as:

$$PC_i = \sum_{j=1}^{m} a_{ij} X_j \; with \, 1 \leq i \leq m \tag{5.1}$$

where X_js are the original variables, m is the number of variables and a_{ij}s are coefficients expressing the weights that the j-th original variables has on the i-th principal component. With this technique, once the m PCs are obtained, a subset of them that containing as much variance as possible is selected. The chosen PCs will be the independent variables of the regression model.

The second technique, i.e., the stepwise regression, considers an initial partial set of variables (the predictors) and then attempts to add a new variable that improves the model (it performs some statistical tests to measure the improvement) or eliminate a variable which coefficient is not statistically significant; this leads to a minimum subset of variables that provide the best model (it yields a sub-optimal solution, since it is a heuristic technique).

Table 5.2: Regressione Models and their Predictive Power

ODC Type	**Technique**	**# of Variables**	R^2	**Adjusted** R^2	**F-test**
Assignment	StepWise	3	0.9884	0.9797	113.92, p = 0.00024
Checking	PCA	5(99.75%)	0.9888	0.9608	34.49, p=0.02841
Interface	PCA	6(99.98%)	0.9997	0.9979	673, p = 0.02948
Algorithm	PCA	6(99.98%)	0.9995	0.9965	377.36, p = 0.03938
Function	StepWise	3	0.9899	0.9823	130.92, p = 00018

While the first technique has the advantage of using totally uncorrelated variables for the model (that is not the case of the stepwise regression), a problem is that the principal components obtained from equation (5.1) do not have a physical meaning; they are a combination of original variables. The stepwise regression, on the other hand, does not use totally uncorrelated variables, but it preserves the physical meaning of predictors.

In our analysis, we used both techniques, computing and evaluating the resulting models and choosing the best one.

5.1.3 Data Analysis

To compare models obtained with the PCA technique and with the stepwise regression, we have first carried out the PC transformation, obtaining 20 principal components from the 20 original metrics. As for PCA, models have been constructed with a number of principal components able to explain at least the 95% of the original data as independent variables. Five regression models have been built, one for each of the considered ODC fault types. Table 5.2 reports the statistics obtained for these models.

In the second column the technique that has been chosen for the model is shown. The third column reports the number of variables used (for PCA, the percentage of explained variance is also reported); in the fourth column the R^2 value is reported, which is the ratio of the regression sum of squares to the total sum of squares. A larger value indicates a better *predictive power*, since it means that more variability is explained by the model with respect to the total variability. The fifth column reports the *adjusted* R^2 value, that accounts for the degree of freedom of the independent variables and the sample population; it is a measure of the robustness of the model.

Then, in the sixth column, the *F-test* value is reported; it tests the null hypothesis that all regression coefficients are zero at a given significance level. Once we built models by using both PCA and stepwise regression, we chosen the best one firstly using this *F-test* value: if a model obtained with either PCA or stepwise was not significant at least at $\alpha = 0.05$, we discarded it and evaluated the alternative technique. If both were significant, we evaluated the R^2 and adjusted R^2 values to choose the model.

Finally, we obtained regression models for all the ODC categories. Observing the statistics, it is worth noting that regression models built by using the PCA technique have the highest $p - values$, an order of magnitude greater than the models obtained with the stepwise technique (a higher $p - value$ means less confidence in the model). However, also in these cases, such values are lower than 0.05 significance level (that is a good value to consider the model as acceptable). R^2 values are also high in all the cases, meaning a high variability explained by the models.

Even if results allow to obtain good predictions, increasing the number of samples, by evaluating more software applications with known ODC fault types, can allow in the future to obtain even more reliable prediction models; this study is a first step toward this direction. Adopting these models, engineers can characterize their software application by using the described metrics as predictors. Charactering applications from the tester's point of view is a relevant piece of knowledge that engineers should have for planning a good verification strategy. This allows them to obtain a preliminary estimate of what types of fault more likely affect the system and can select verification techniques more prone to cope with them.

5.2 Software Aging Analysis

The same principle as the study presented in the previous section inspired the analysis of software aging presented in the following. We recall that the software aging is a phenomenon due to the continued and growing degradation of software internal state caused by accrued error conditions (such as round-off errors or a wrong management of system's resources). A typical example is an unreleased memory region inside a program's heap area, which cause a memory depletion trend that eventually leads to crash. Common aging indicators are the throughput loss, indicating the decrease of performances that a system affected by aging undergoes, and the memory depletion, i.e., the quantity of memory lost per unit time due to aging. In the following we refer to memory depletion as aging indicator (that is the most commonly adopted).

As discussed in section 1.1.3 and 2.3, this is typically a long running phenomenon coped with at operational time. However, we intend to address it already in the testing phase.

The goal of the presented analysis is to give a preliminary knowledge to engineers responsible for verification of software modules/components/subsystems

more prone to suffer from aging, so that they can intervene on them with a focused testing effort. The hypothesis (to verify) is that the aging phenomenon is related to several software's features that can be described by means of software metrics. In particular, we hypothesize that the software size, its complexity, the usage of some kinds of programming structures related to the resource management, the use of arithmetic operators, and other features is tightly related to the occurrence of aging-related bugs. For instance, it is more likely for a developer to omit a "free" function call (to release memory) in complex and large code than in a relatively simple and small piece of code. As stated, the principle underlying this kind of analysis is the same of fault-proneness models previously applied; however it has not been possible to conduct this analysis together with the previous one, for two main reasons:

- For software aging analysis, we need to choose a wider set of metrics than the 20 adopted previously; other than classical metrics, a subset of specific metrics potentially bound to this phenomenon (i.e., that could be symptomatic of aging bugs) has to be taken into account.

- The software applications used as sample population cannot be the same of the previous analysis, since the response variable is different: here we looked for software suffering from aging as samples of the response variable, whereas in the previous study, applications with a known content of ODC fault types were sought.

Due to additional difficulties we faced during the analysis, we also varied the experimental procedure to obtain statistically relevant results. In the following, we detail the main steps.

5.2.1 Experimental Procedure and Data Analysis

Metrics Selection

In this analysis, an initial set of 51 software metrics was considered. Due to their inter-correlation, subsequent analyses will show that consider just a small subset of them are relevant as predictors. In particular, other than the 20 metrics considered in the previous analysis, the following additional metrics are collected:

Code Volume,

Logical source lines of code (SLOC-L), Physical source lines of code, Number of comment words (CWORDS).

Files Volume

Source Files, Complexity, McCabe Cyclomatic Complexity.

Halstead's Metrics (science metrics)

These metrics are also known as Software Science Metrics. They are useful to describe the software complexity in terms of operands and operators. We considered their Mean and Variance (as for instance the mean Volume, Length, Bugs Delivered, Vocabulary, Difficulty, Effort).

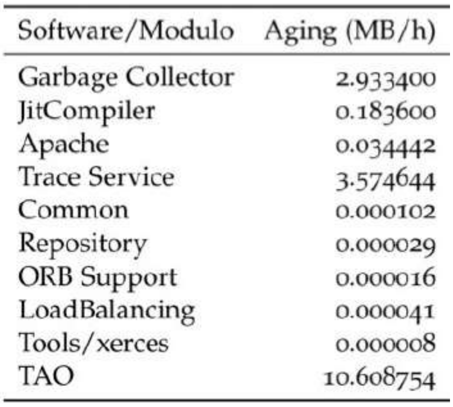
Resource Management Metrics

These metrics are take into account the usage of system's resources: a number of memory allocation much higher than memory de-allocations in a software, could indicate a bad memory management, and hence potential source of aging; similarly for other resources that could be used and never released. In this study, we considered:

Number of memory allocations, Number of memory de-allocations, Number of opened files, Number of closed files

Software Selection and Metrics Extraction

To select the sample population, we looked for software systems already known as suffering from aging. They are showed in Figure 5-3.



Software/Modulo	Aging (MB/h)
Garbage Collector	2.933400
JitCompiler	0.183600
Apache	0.034442
Trace Service	3.574644
Common	0.000102
Repository	0.000029
ORB Support	0.000016
LoadBalancing	0.000041
Tools/xerces	0.000008
TAO	10.608754

Figure 5.3: Software selected for the aging analysis

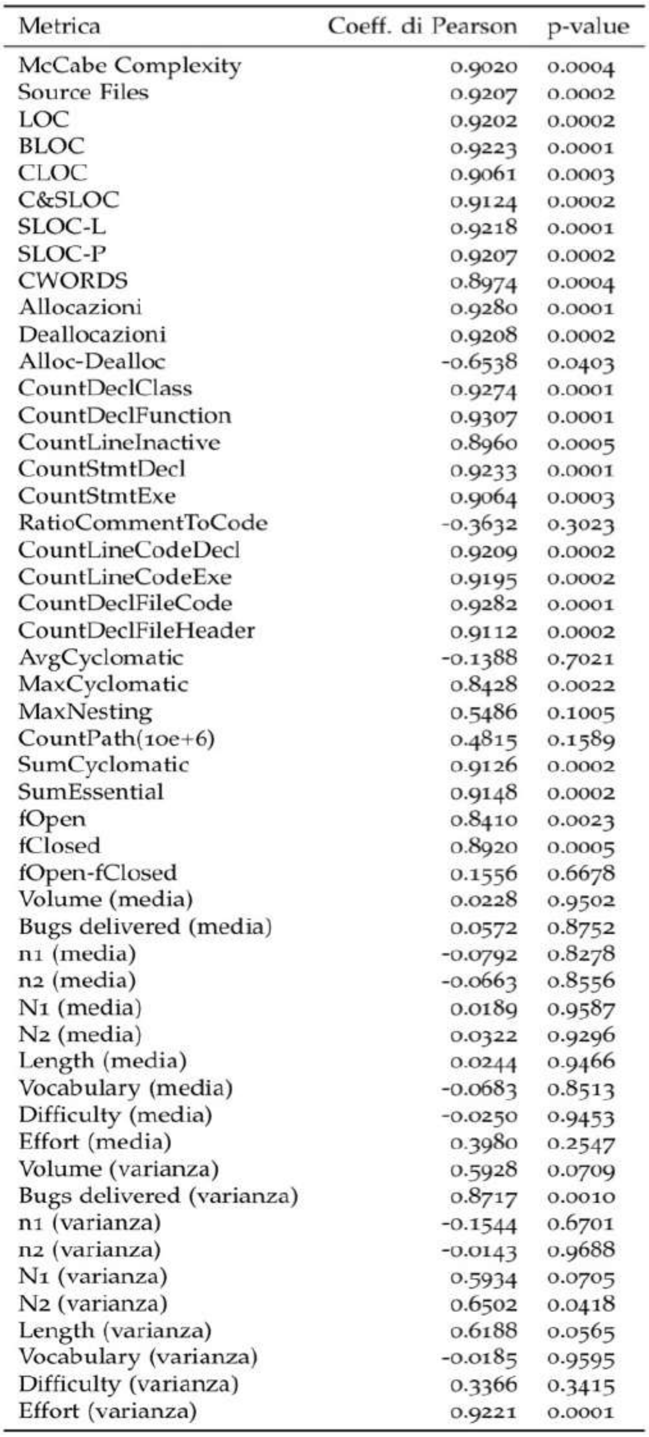
The first two software modules belong the Java Virtual Machine (JVM): authors in [16], conducted a deep analysis of software aging inside the Java Virtual Machine, founding out a notable aging contribution in both *Garbage Collector* module, and in the *Just-in-Time Compiler* module. *Apache* is the notorious Web Server, which has been subject of several studies about software aging, as reported in chapter 2, reporting that it is affected by this phenomenon. The successive six components are software modules belonging to a CORBA-based middleware platform, named CARDAMOM, developed by Thales and Selex-SI, and designed to support the development of software architectures for safety and mission critical systems. The study in [112] showed that these components are affected by aging. The same studies highlighted the presence of software aging also in a well-known C++ implementation of a CORBA-compliant ORB, that is The Ace ORB (TAO). Such software systems were used a sample population to derive regression models using the described metrics as predictors.

Verifying the presence of correlation

Before proceeding with the analysis, the presence of significant correlations has been verified, as in the previous analysis. Results, showed in Figure 5-4, allow stating that there is a correlation statistically significant between many metrics and the aging phenomenon; so the next step of the analysis can take place.

Subset Selection and Classification

Observing aging trends of the selected software applications, it is evident the difference between two groups of applications, the former (composed of the Garbage Collector, Jit Compiler, Apache, Trace Service and TAO) exhibiting an aging trend of orders of magnitude greater than the latter (the remaining five modules). Thus, the first regression model we built based on this data presented an excessive standard error (amounting to 1.6 MB/h). We therefore proceeded with a features selection and classification step, in order to arrange data according to these two groups (named BigAging and LittleAging). In particular, we sought those software metrics (i.e., the features) most able to establish if an instance belongs to a class or another. We adopted a simple algorithm that provides significance indexes for

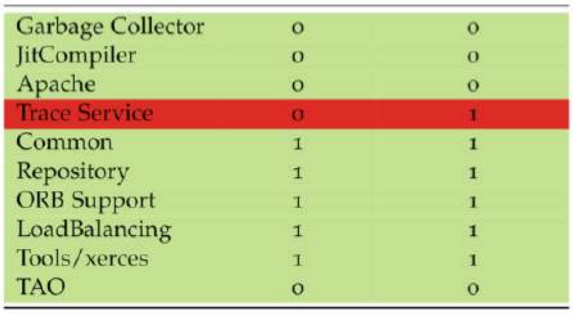


Metrica	Coeff. di Pearson	p-value
McCabe Complexity	0.9020	0.0004
Source Files	0.9207	0.0002
LOC	0.9202	0.0002
BLOC	0.9223	0.0001
CLOC	0.9061	0.0003
C&SLOC	0.9124	0.0002
SLOC-L	0.9218	0.0001
SLOC-P	0.9207	0.0002
CWORDS	0.8974	0.0004
Allocazioni	0.9280	0.0001
Deallocazioni	0.9208	0.0002
Alloc-Dealloc	-0.6538	0.0403
CountDeclClass	0.9274	0.0001
CountDeclFunction	0.9307	0.0001
CountLineInactive	0.8960	0.0005
CountStmtDecl	0.9233	0.0001
CountStmtExe	0.9064	0.0003
RatioCommentToCode	-0.3632	0.3023
CountLineCodeDecl	0.9209	0.0002
CountLineCodeExe	0.9195	0.0002
CountDeclFileCode	0.9282	0.0001
CountDeclFileHeader	0.9112	0.0002
AvgCyclomatic	-0.1388	0.7021
MaxCyclomatic	0.8428	0.0022
MaxNesting	0.5486	0.1005
CountPath(10e+6)	0.4815	0.1589
SumCyclomatic	0.9126	0.0002
SumEssential	0.9148	0.0002
fOpen	0.8410	0.0023
fClosed	0.8920	0.0005
fOpen-fClosed	0.1556	0.6678
Volume (media)	0.0228	0.9502
Bugs delivered (media)	0.0572	0.8752
n1 (media)	-0.0792	0.8278
n2 (media)	-0.0663	0.8556
N1 (media)	0.0189	0.9587
N2 (media)	0.0322	0.9296
Length (media)	0.0244	0.9466
Vocabulary (media)	-0.0683	0.8513
Difficulty (media)	-0.0250	0.9453
Effort (media)	0.3980	0.2547
Volume (varianza)	0.5928	0.0709
Bugs delivered (varianza)	0.8717	0.0010
n1 (varianza)	-0.1544	0.6701
n2 (varianza)	-0.0143	0.9688
N1 (varianza)	0.5934	0.0705
N2 (varianza)	0.6502	0.0418
Length (varianza)	0.6188	0.0565
Vocabulary (varianza)	-0.0185	0.9595
Difficulty (varianza)	0.3366	0.3415
Effort (varianza)	0.9221	0.0001

Figure 5.4: Metrics correlation with software aging

each feature representing their ability to classify instances, named IndFeat (i.e., independent significance features test). The most significant metrics turned out to be Halstead's Metrics, and in particular: *the Mean Volume, the Mean Effort, the Variance of Volume, the Variance of N1, the Variance of N2 and the Variance of Length.*

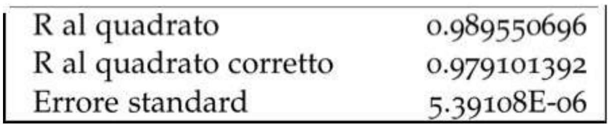
Such metrics were then used to train a simple classifier (implemented by means of the MATLAB libSVM library). The classifier has been validated by the leave-one-out cross-validation method, which consists in building a classifier with *n-1* software samples, using the remaining sample as test set, and iterating the procedure changing the test set each time (i.e., n steps). Hence, *n-1* software applications are used as training set, while the remaining one as test set. Table 5-5 shows that an error of 10% is committed with the selected metrics (0 represents the BigAging group, 1 the littleAging).



Garbage Collector	0	0
JitCompiler	0	0
Apache	0	0
Trace Service	0	1
Common	1	1
Repository	1	1
ORB Support	1	1
LoadBalancing	1	1
Tools/xerces	1	1
TAO	0	0

Figure 5.5: Classification error

However, since in several experiments the software Trace Service has always been the source of error, we decided to remove it and repeat the validation. In this latter case the error with the leave-one-out cross-validation was null. Once built the classifier, for each class a distinct regression model was built.



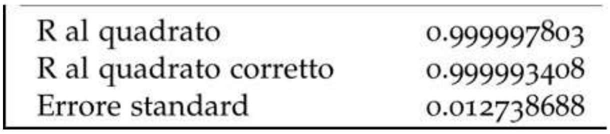
R al quadrato	0.989550696
R al quadrato corretto	0.979101392
Errore standard	5.39108E-06

Figure 5.6: Statistics for the LittleAging group

Building Regression Models

Regression models were formulated, as in the previous analysis, with both the PCA technique and the stepwise procedure. The stepwise procedure produced better models for both groups; hence we report here only these results. As for the group LittleAging, the stepwise procedure selected two metrics (CountLineInactive and RatioCommentToCode) as predictors (at 95% confidence level), providing a model which statistics are reported in Figure 5-6. The mean relative error with this model amounted to 11.49%.

As for the group BigAging, the stepwise procedure pinpointed the following two metrics: Lines of Code and the Mean Volume. Statistics are reported in Table 5-7:



R al quadrato	0.999997803
R al quadrato corretto	0.999993408
Errore standard	0.012738688

Figure 5.7: Statistics for the BigAging group

The mean relative error with this model amounted to 6.81%. Obtained models show therefore good predictive power inside each class. Model assumptions have been verified as in this case of the empirical analysis of chapter 4 and in the previous analysis in this chapter; hence adopting the same tests to verify the omoschedasticity, the normality of residuals and the non-correlation between independent variables and residuals.

In summary, differently from the ODC fault type analysis, in this case an engineer that wants to obtain a prediction of software aging of a given software module has to carry out the following steps: (i) to extract the metrics with an automatic tool; (ii) then s/he has to apply the classifier to determine if its software belongs to the *LittleAging* group or to the *BigAging* group; (iii) finally s/he has to apply one of the two described regression models according to class it belongs to. In this way a preliminary estimate of aging trend in the testing phase is obtained starting from software metrics. Of course, standing the validity of these results, further samples would allow for improving these characterizations; hence all the presented empirical analyses, from chapter 4 and 5, leave wide room for improvements of their predictions. Chapter 6 discusses such aspects, where a procedure to self-enrich such results is proposed, with the goal of providing, with time, more and more reliable and accurate results.

Chapter 6

Steps for Reliability-Oriented Verification

This chapter intends to synthesize what exposed in the previous chapters in the definition of a step-by-step procedure that critical systems developer could consider for carrying out verification activities. The approaches proposed for efforts allocation, software system characterization and verification techniques selection are logically placed in a procedure conceived to exploit at their best the presented results. Moreover, the chapter describes how such steps should be incorporated in a verification process in a self-refining way; i.e., in order that results of each process application iteratively enrich the empirical data on which the procedure is based. Indeed, being the procedure steps based on empirical data, its iterative adoption across several projects allows to enrich the empirical base of knowledge, increasing the confidence in the suggested policies and yielding higher and higher accuracy across the projects.

6.1 Step-by-step procedure outline

The main issue arising from what presented in chapter 3, 4 and 5 is how tester can exploit at the best those results. Although there is no a unique right answer to this, in the following a possible procedure is proposed.

Steps of this procedure describe actions that a tester should take in order to pursue an effective verification activity.

The procedure starts by adopting the approach described in chapter 3 for the effective allocation of verification effort aimed at assuring a given level of reliability. Depending on the adopted solution (i.e., basic, extended or complete), engineers distribute the effort to the various subsystems/components. For each of them a testing session is carried out. Note that effort allocation step may not always apply, as discussed in section 6.1.1; in this case the testing session (and the next steps) refers to the entire system rather than on single parts of it.

In order to select the best techniques for that specific subsystem (or system), a set of preliminary actions has to be taken before starting the testing session. We gathered such actions in the **"Initialization"** step. They include the selection of the first technique, and the system characterization in terms of ODC-faults content and expected aging trend (by means of proneness models presented in chapter 5).

Aging is considered separately; since aging bugs can be anyone of the ODC faults. Instead of the compared techniques, aging bugs need to be treated with analysis techniques (dynamic analysis is preferable). However, there is still a little work on aging-oriented testing; dynamic analysis tools, such as Valgrind, have successfully been adopted [112].

For the purpose of our work, we simply have to indicate what part of the system may be more or less affected by aging; the tester will then devote greater effort for aging detection (likely through analysis) to such parts.

Once these preliminary steps are completed, the testing session execution starts, by adopting the technique selected in the first step. As the testing proceeds, faults are detected and removed. The number of faults detected per unit time will decrease over time, since a technique (whatever it is) always adopts the same criterion to stress the code; as the faults that it is able to detect are removed, its detection ability decreases. When this technique becomes ineffective (section 6.1.2 will detail what “ineffective” means), another technique has to be adopted. A *techniques selection* step will then be responsible for choosing the next technique most suited for that system, based on the ODC faults prediction. This choice is made dynamically, since as faults are removed during testing, the ODC faults content varies. This is described in section 6.1.3. Techniques selection is of course repeated when the same ineffectiveness condition, as with the first technique, occurs, until the available testing time for that system/subsystem expires. Figure 6.1 summarizes the outlined procedure. The

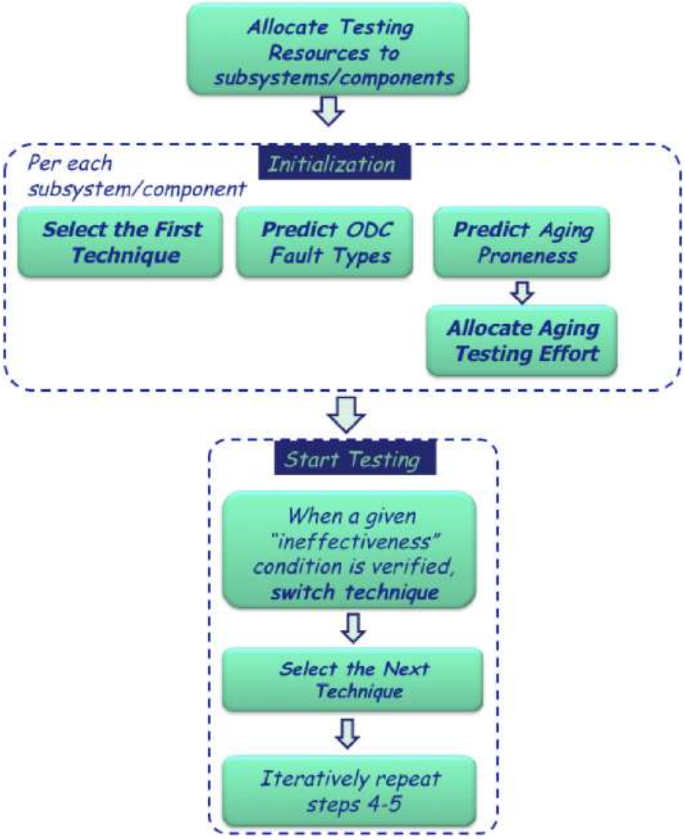

Allocate Testing Resources to subsystems/components
Initialization
Per each subsystem/component
Select the First Technique
Predict ODC Fault Types
Predict Aging Proneness
Allocate Aging Testing Effort
Start Testing
When a given "ineffectiveness" condition is verified, switch technique
Select the Next Technique
Iteratively repeat steps 4-5

Figure 6.1: Step-by-step Procedure

following subsections describe the various steps in more detail.

6.1.1 Effort Allocation

The first step is the allocation of available resources for verification to parts of the system, by adopting the model described in chapter 3. Hence, engineers set a reliability level to achieve, and gather all the necessary information to apply one of the three solutions: basic, extended, and complete, according to what they are able to know about the system and its components. Information gathering is discussed in section 3.4. Recall that the best case is when a reliability growth model for each component can be obtained (along with the necessary architectural information), which lead to a complete solution, i.e., a solution where the exact testing efforts are assigned.

An important remark is that the number of components of the system is an analysis choice; the granularity is chosen depending on the needs. In the case of experiments presented in section 3.5, by choosing three big components (rather than many small components) it has been more relevant to apply the approach to distribute testing efforts: indeed, when components are large, it is more likely to have significant differences in the required testing times, whereas with many small components, avoiding the usage of the model and devoting the same time to each components, would lead to slighter errors. With few large components, assigning the same testing time to each component could instead lead to significant errors. It becomes in this case important to assign the right testing effort to the components. Moreover, another reason to prefer large components, especially when the user wants to adopt a “complete solution” (i.e., with SRGMs), is that with small components few failure data are available to build an accurate reliability growth model. The granularity of components to be selected for a useful analysis also depends on how much they are decoupled. High decoupling indicates that components can be more independently tested than with coupled components: this enables the possibility to better schedule testing activities and organize the testing team to work on different components, by using results obtained by the allocation model.

These considerations for components granularity also indicate that this step may not always take place. If engineers deal with a very small system, or with few small components, allocating efforts according to the model may yield a very little benefit. It may happen that all the testing team is preferred to work on the system as a whole, especially when the system is small and with highly coupled components. In this case, engineers can prefer to skip this step and proceed with the next one, applied to the entire system rather than on single subsystems/components.

6.1.2 Initialization

Steps from initialization to the end of procedure, can be applied either to a subsystem/component to which a given testing effort has been allocated, or to the entire system, in the case the previous step did not apply. Hence, in the following we refer to system or to subsystem without distinction, unless differently specified. Initialization consists of four main activities:

- **Selecting the first technique to apply**. Since reliability improvement is the main objective, the technique to adopt should be the one most prone to improve reliability. In chapter 4, we compared techniques also with respect to the delivered reliability (section 4.4); results showed that the best performance was yield by the statistical testing and by the functional testing (without a statistically significant difference). The main reason for this, is that these "Demonstrative" techniques (as we called them in chapter 4), aim to remove high-occurrence faults and rapidly achieve a notable level of reliability. However, as discussed in section 4.5, further improvement requires other techniques to be adopted, capable of detecting faults with lower occurrence probability. *How* to select these techniques and *when* is the right moment to switch technique is described in the next section. In this step, based on results of chapter 4, either *statistical* or *functional* testing can be adopted as first technique.

- **System characterization in terms of fault types and content**. Results of chapter 5 allow to give a preliminary characterization of the software system in terms of fault types (according to the ODC scheme). Starting from software metrics, which describe the features of the software, regression models provide the estimates of the content of faults per each ODC type. Software features are therefore used as predictors of faults content. This characterization is useful at runtime, when a new technique has to be selected, since the choice will also depend on what types of fault are expected to be present. In this sense, techniques selection is "adaptive", i.e., it is tailored to specific system being tested, to its expected fault types and hence to its features.
- **System characterization in terms of expected aging trend**. Regression models for aging prediction (presented in chapter 5) are used in this initialization step to characterize the system in terms of expected aging trend. Aging can be due to different

types of ODC faults. Predictions can be made on the entire system/subsystem, or also to software modules of several granularity within a system/subsystem.

- **Aging-testing effort allocation**. Aging-oriented testing is not yet an established topic in the literature; it is generally detected with dynamic analysis techniques. Hence the previous characterization is used, in this step, to understand which subsystem/modules/components needs more attention. Without determining the technique to be applied (let us suppose it is a dynamic analysis technique), this step allocates how much effort has to be devoted to various modules, specifically for aging detection purposes. Hence the next steps (dealing with techniques selection) do not apply to aging.

6.1.3 Switching techniques

When the steps required by initialization are complete, the testing session actually begins. As already discussed, to achieve high reliability in less time, the first technique should be replaced after some time, when its "detection power" is exhausted. Initially, faults more prone to be detected by the technique are revealed and removed; as testing proceeds, less and less faults are detected, since the technique tries to look for faults always with the same criterion, probably addressing always the same fault types and in near regions of the code. At a given point it should be replaced. However, the first question is *when*?

Based on the observation of the detection trend, engineers can decide to switch to another technique when the **detection rate** of the current technique (i.e., the number of faults detected per unit time) goes under a given threshold.Although several different measures can be adopted to evaluate the current “ineffectiveness” of a technique, detection rate is the most suited to describe the problem outlined.

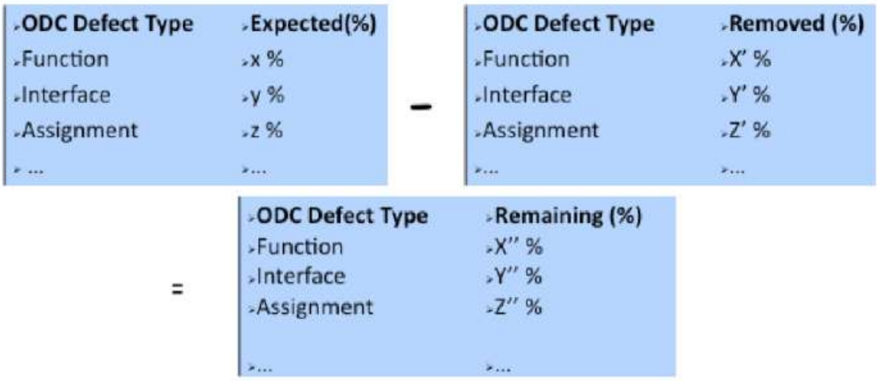
The threshold should be set depending on the available testing time, or on the ratio between the expected number of faults to be still removed and the remaining testing time. For instance, if at time t, engineers evaluate that in the remaining testing time, with the current detection rate, only a small fraction of faults are removed (that does not meet their expectation), technique has to be changed. Once decided when to switch technique, the next step is to establish *what* is the next technique.

Techniques selection has to be carried out based on the prediction of ODC faults content. The study conducted in chapter 4 also investigated the impact of ODC fault types on detection ability of techniques. Hence if, for example, a technique is more prone to detect the *Interface* faults, and ODC faults prediction estimates a high number of *Interface* faults expected to be present, than that technique could be selected. However also in this case, there is no an exact solution, but some heuristics that can be adopted (as for the threshold choice).

More formally, given the initial prediction of ODC fault types and considering the faults removed until time t (i.e., the time to switch technique), an estimate of residual ODC fault types at time t is obtained. Figure 6-2, summarizes the concept. Engineers know how many faults of what type are expected to be present at time t.

Based on this result, the next technique can be:

- Simply the one more prone (in our experiments) to detect the predominant faults



ODC Defect Type	Expected(%)
Function	x %
Interface	y %
Assignment	z %
...	...

−

ODC Defect Type	Removed (%)
Function	X' %
Interface	Y' %
Assignment	Z' %
...	...

=

ODC Defect Type	Remaining (%)
Function	X'' %
Interface	Y'' %
Assignment	Z'' %
...	...

Figure 6.2: Removal of ODC Fault types durig testing

type.

- The one that in the average allows detecting more faults among the remaining ones. Saying that a technique is more prone to remove *Interface* faults does not mean that it will detect only *Interface* faults. Hence, the percentages reported in the Figure 4.2 in chapter 4, showing the relative effectiveness of techniques with respect to fault types, could be considered to define an average index. For instance, if functional testing detects 80% for *Assignment* category and 40% for the *Interface* category, while robustness testing detects 30% of *Assignment* and 60% of *Interface*, a simple index can be: *Functional = #Assignment * 80% + #Interface * 40%, and Robustness = #Assignment*40% + #Interface*60%.* Technique with higher index will be the next one.

- Even though faults detection ability and reliability improvement showed in the average a similar trend, results of chapter 4 also highlighted that this is not always true.

> Hence a tailored index for reliability can be formulated, in order to choose the technique that, in the presence of those specific kinds of faults, showed higher reliability. Observing results, which report reliability achieved under different initial conditions (e.g., with different initial faults type percentages) several indexes can be formulated. One suitable index could be the observed increment in the average reliability when the fault of type j varied from $PERCENTAGEmin$ to $PERCENTAGEmax$, i.e., the increment of reliability per percentage unit relatively to the fault type j. Alternatively, indexes derived from the Analysis of Variance (ANOVA) can be computed. Adopting ANOVA techniques would allow to relate each ODC category to reliability to infer how the variation of ODC type impacts on reliability.

The final step is the iteration of the *techniques selection*. Indeed, once a given technique is selected, its detection ability, after some time, will decrease; hence a new technique selection has to take place, until the available testing time expires.

It is finally worth to note that, alternatively to the described procedure, techniques can be selected before starting the testing phase, based only on the initial fault types distribution (rating the techniques more prone to cope with that faults distribution, and apply them subsequently). This static solution is simpler to implement; however, it does not allow to schedule the order of application of techniques dynamically, based on current faults type distribution and hence yields worse results.

6.2 Improving the Process

This final section intends to show how the outlined procedure can be embedded in a verification process.

Steps described in the previous sections are intended to exploit in the best possible way results of the analyses and of the approaches proposed in the thesis. One commonality of such approaches is that they are based on empirical data. The allocation model presented in chapter 3 needs data to train the model parameters (for both the architectural model and for SRGMs of components); the system characterization and techniques selection presented in chapter 4 and 5 are entirely based on empirical observations, produced by massive experimental campaigns. The effectiveness of a verification activity conducted by means of the outlined procedure will mainly depend on the accuracy of the empirical-based predictions. Hence, a natural way to improve the effectiveness is to exploit results produced in one project to train iteratively the predictive models, obtaining progressively more accurate results.

The base of knowledge produced by a company can in this way be exploited to increase the confidence in the suggested policies, and to set up more and more reliable verification plans. In particular, the following data are worth to be gathered:

- **Inter-failure times** of each system's component. These are useful to build Software Reliability Growth Models of components, especially useful if the same components are used across several projects (that is a widely established practice). Considering as components not only built-in-house applicative components, but also off-the-shelf items ranging from the applicative layer to the middleware and even to the operating system layer, it is clear that their reuse is a very frequent situation. This improves SRGMs accuracy.

- **Detected faults and adopted techniques**. Recording the percentages of faults detected by each technique and the type of faults classified according to the ODC, is useful to improve the confidence levels of comparisons among techniques, as the one conducted in the study of chapter 4, and their relationships with fault types.
- **ODC Faults Type-Metrics**. The type of ODC faults actually present in each software modules, along with the corresponding software metrics, allow to improve the predictive power of regression models presented in chapter 5.
- **Aging Trend-Metrics**. Monitoring at runtime the aging trends of software modules and extracting the corresponding metrics, allows enriching the empirical models for aging prediction. Due to the poor body of studies about aging, having software instances with known aging estimates is essential to improve the empirical analysis with a richer sample population.

Chapter 7

Conclusion

Companies developing critical systems often encounter serious difficulties in satisfying reliability requirements, in many cases imposed by certification standards, at competitive and acceptable cost and time. Since most of development cost is due to the verification phase, carrying out an effective verification is of paramount importance to significantly reduce the overall cost.

This dissertation focused on software verification effectiveness of large critical software systems. Specifically, the problem addressed in this thesis is how engineers can plan an effective verification activity oriented to improve the final system reliability. Starting from the analysis of the current state-of-the-art about verification strategies in this field, the author identified the main open challenges to be faced, by trying to figure out what are the most crucial steps that engineers need to take for an effective planning.

In particular, the identified needs include: (i) the suitable allocation of efforts available for verification to the system's components, (ii) the scheduling of the most appropriate verification technique(s), specifically oriented to improve reliability and tailored for the system being tested, and (iii) the handling of phenomena usually accounted for only at runtime, but that in a critical system cannot be disregarded (e.g., software aging). Such activities, although crucial for the final results, are often left to the engineers' intuition and experience, due to the lack of convincing approaches coping with them. The present dissertation focused on these challenges, and proposed a solution, based on objective criteria rather than intuition, to achieve high effective verification processes in the considered class of systems. Its support to a reliability-oriented effective verification process in the most critical activities is the major contribution. In particular, for the investigated activities and the identified need, the thesis contributed by proposing:

- **An optimization model** to allocate the verification resources to different system components in order for the system to achieve a required reliability level at minimum verification costs. The purpose of the model, through the tool implementing it, is to drive engineers in the first phase of verification activities, when efforts have to be allocated. An architecture-based model (in particular a DTMC) was used to describe the software architecture. The optimization model was conceived for providing flexible solutions, at different levels of detail, according to the information provided by the user. This allows engineers to actually adopt an allocation model even with a limited knowledge about the system and its components, in contrast to past models. The solution considers the OS as a component and the potential fault tolerance mechanisms that a component could employ, being closer to the realty of critical systems. Experiments showed that the prediction ability of the model and its sensitivity to the variation of potential sources of errors are widely acceptable. As side effect, the adopted architecture-based solution allows to obtain information about the sensitivity of some components and their impact on the final reliability that are useful to successive designs for future versions.

- **Empirical analyses** to investigate some relevant aspects of the verification activity of software systems. Experimental campaign results provide a valuable support to engineers for properly planning the verification strategy. Results of such analyses have been then used (i) to ease the selection of the most appropriate techniques for improving reliability, i.e., the ones most suited to the specific features of the software under test, and (ii) to dealing with the phenomenon of software aging, by providing estimates of aging trends potentially affecting software modules. Empirical analyses allowed to tune prediction models able to characterize the system, and to characterize the behaviour of verification techniques with respect to several relevant factors, such as the types of faults, related to the system's features. Results are quite useful, in that they quantitatively support engineers' intuition or experience in judging what is the best technique for their case.

- The work defines **a procedure to improve verification processes** in the considered class of systems, able to iteratively refine results across the developed projects. By following this procedure, which exploits results of the outlined contributions, testers may determine the most suited set of techniques, their best order of application and the final confidence at which reliability is improved. Its iterative adoption (i.e., its inclusion in the verification process) across more projects allows enriching the empirical base of knowledge, increasing the confidence in the suggested policies. The continuous refinement would yield higher quality and more confident processes where resources, approaches and techniques systematically serve the final reliability objective in a cost-effective way.

Finally, it is worth to remark that the solution proposed in this thesis supports verification process effectiveness independently from organizational factors, and may benefit critical systems companies adopting various different development cycles, personnel management policies and bound to different certification standards. In other words, it has been conceived to provide an added value to the quality assurance process independently from organizational settings.

Bibliography

[1] M. Pezze, M. Young, Software Testing and Analysis: Process, Principles and Techniques, Wiley, John & Sons, Incorporated, (January 2007)

[2] T., Pasquale, E., Rosaria, M., Pietro, O., Antonio: Hazard analysis of complex distributed railway systems, in proc. of *the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS'03)*, pp. 283-292, Oct. 2003.

[3] Functional safety and IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems *Produced by IEC/SC65A/WG14, the working group responsible for guidance on IEC 61508.*, Sep. 2005.

[4] DO-178B/ED12B, "Software consideration in airborne systems and equipment certification," RTCA and EUROCAE, Dec. 1992.

[5] K. Fowler: Mission-Critical and Safety-Critical Development, in *IEEE Instrumentation and Measurement Magazine*, Dec. 2004

[6] Y. Huang, C.M.R. Kintala, N. Kolettis, and Fulton N.D. Software rejuvenation: Analysis, module and applications. *In Proc. of the 25th International Sym- posium on Fault-Tolerant Computing* (FTCS-25), Pasadena,CA (USA), pp. 381–390, 1995.

[7] E. Marshall. Fatal Error: How Patriot Overlooked a Scud. Science, 255:1347, 1992.

[8] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on De- pendable and Secure Computing*, 1(1):11-33, 2004.

[9] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 90.1 (Part Number 33579), Tandem Computers Inc., January 1990.

[10] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73-83, 1997.

[11] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, M.-Y. Wong, Orthogonal Defect Classification-A Concept for In-Process Measurements, *IEEE Transactions on Software Engineering*, Vol. 18, No. 11 (November 1992), 943-956.

[12] Joao Duraes and Henrique Madeira. Definition of software fault emulation operators: A field data study. *In Proc. of the 2003 International Conference on Dependable Systems and Networks* (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, pages 105-114. IEEE Computer Society, 2003.

[13] J.A. Duraes, H.S. Madeira, Emulation of Software Faults: A Field Data Study and a Practical Approach, Software Engineering, *IEEE Transactions on Software Engineering*, Vol.32, No.11, Nov. 2006, 849–867

[14] J. Gray. Why do computers stop and what can be done about it? *In Proc. of the Symposium on Reliability in Distributed Software and Database Systems* (SRDS), Los Angeles, USA, pages 3-12, 1986.

[15] Michael Grottke and Kishor S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer Communications*, 40(2):107-109, 2007.

[16] Software Aging Analysis Of Off The Shelf Software Items, Salvatore Orlando, Ph.D Thesis, 2007.

[17] K.S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications," John Wiley and Sons, 2001.

[18] R. Mullen. The Lognormal Distribution of Software Failure Rates: Origin and Evidence. *Proc. of the 9th IEEE International Symposium on Software Reliability Engineering*, November 1998.

[19] R. A. Sahner, K. S. Trivedi, and A. Puliafito. Performance and Reliability analysis of Computer Systems; An Example-based Approach Using the SHARPE Software Package. Kluwer Academic Publisher, 1996.

[20] A. P. Wood. Multistate block diagrams and fault trees. IEEE Transactions on Reliability, R-34:236-240, 1985.

[21] W. G. Schneeweiss. Petri Nets for Reliability Modeling. LiLoLe Verlag, 1999.

[22] J. Bechta-Dugan, S. J. Bavuso and M. A. Boyd, Dynamic fault-tree models for fault-tolerant computer systems, IEEE Transactions on Reliability, 41, pp.363-377, 1992.

[23] A. Bobbio, G. Franceschinis, R. Gaeta and L. Portinale, ÒParametric Fault-Tree for the Dependability Analysis of Redundant Systems and its High Level Petri Net SemanticsÓ, IEEE Transactions Software Engineering, 29 (270-287) 2003.

[24] D. Codetta Raiteri, G. Franceschini, M. Iacono and V. Vittorini, Repairable Fault Tree for the automatic evaluation of repair policies, in*International Conference on Dependable Systems and Networks* (DSN '04), (Florence, Italy), pp.659-668, IEEE Computer Society, 2004.

[25] C. B. Almeida and K. Kanoun, Construction and Stepwise Refinement of Dependability Models, *Performance Evaluation*, vol. 56, 277-306, 2004.

[26] Y. Dai, Y. Pan, X. Zou, A Hierarchical Modeling and Analysis for Grid Service Reliability, *IEEE Trans. on Computers*, vol. 56, 681-691, 2007.

[27] Trivedi, K. Wang, D. Hunt, D.J. Rindos, A. Smith, W.E. Vashaw, B., Availability Modeling of SIP Protocol on IBM©WebSphere ©, *Proc. of the 14th IEEE Pacific Rim Intl. Symposium on Dependable Computing*, 2008, 323-330.

[28] G. A. Hoffmann, K. S. Trivedi , M. Malek, A Best Practice Guide to Resource Forecasting for the Apache Webserver, *Proc. of the 12th IEEE Pacific Rim Intl. Symposium on Dependable Computing*, 2006,183-193.

[29] W. E. Smith, K. S. Trivedi, L. A. Tomek, J. Ackaret, Availability analysis of blade server systems, *Ibm Systems Journal*, vol. 47, no. 4, 2008.

[30] G. Ciardo and K. S. Trivedi, Decomposition Approach to Stochastic Reward Net Models, *Performance Evaluation*, vol. 18, 37-59, 1993.

[31] D. Daly, W. H. Sanders, A connection formalism for the solution of large and stiff models, *34th Annual Simulation Symposium*, 2001, 258-265.

[32] Garzia, M.R., Assessing the Reliability of Windows Servers, *Proc. of Dependable Systems and Networks*, (DSN-2002).

[33] Silva, G.J., Madeira, H., Experimental dependability evaluation. In Diab, H.B., Zomaya, A.Y., *eds.: Dependable Computing Systems: Paradigms, Performance Issues, and Applications.* Wiley (2005) 319-347.

[34] D. Tang, R.K. Iyer, Dependability Measurement and Modeling of a Multicomputer System, *IEEE Trans. on Computers*, 42(1), 62-75, 1993

[35] D.Long, A.Muir, R.Golding, A Longitudinal Survey of Internet Host Reliability, *Proc. of the 14th Symposium on Reliable Distributed Systems.*

[36] Kesari Mishra, K.S. Trivedi, Model Based Approach for Autonomic Availability Management, *Proc. of the Intl. Service Availability Symposium, Helsinki* , Finlande, 2006, vol. 4328, 1-16

[37] Haberkorn, M. Trivedi, K., Availability Monitor for a Software Based System, *Proc. of the 10th IEEE High Assurance Systems Engineering Symposium*, 2007. HASE '07, 21-328

[38] Kalyanaraman Vaidyanathan and Kishor S. Trivedi. A comprehensive model for Software Rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2):124-137, 2005.

[39] A.L. Goel, K. Okumoto, A time dependent error detection rate model for software reliability and other performance measures, IEEE Transactions on Reliability, R-28(3) 206-211, 1979

[40] A.L. Goel,Software reliability Models: Assumptions, limitations, and applicability, IEEE transactions on software engineering, SE-11, 1411-1423 (1985)

[41] S. Gokhale, K. Trivedi, Log-logistic software reliability growth model, Proc. of the 3rd IEEE Intl. High Assurance Systems Engineering Symp., 1998

[42] J.D. Musa, K. Okumoto, A logarithmic Poisson execution time models for software reliability mesaurement, Proc of the 7th Intl. Conference on Software Engineering, 1983, 230-237.

[43] S.S. Gokhale, W. E.Wong, J.R. Horganc, K. S. Trivedi, "An analytical approach to architecture-based software performance and reliability prediction," Performance Evaluation, vol. 58, issue 4, pp. 391-412, 2004.

[44] K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi, "Comparison of architecture-based software reliability models," Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE '01), pp. 22-31, 2001.

[45] S. Gokhale, M.R. Lyu, K.S. Trivedi, "Incorporating fault debugging activities into software reliability models: A simulation approach," IEEE Trans. on Reliability, vol. 55, No. 2, pp. 281–292, June 2006.

[46] W.Wang, Y.Wu, M.H. Chen, "An architecture-based software reliability model," Proc. of the Pacific Rim Dependability Symposium, 1999.

[47] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," Performance Evaluation, vol. 45, issue 2-3, pp. 179-204, 2001.

[48] S. Gokhale, K.S. Trivedi, “Time/Structure Based Software Reliability Model,” Annals of Software Engineering, vol. 8, pp. 85-121, 1999.

[49] A. Reibman, K.S.Trivedi, “Numerical transient analysis of Markov models,” Computers & Operations Research, vol. 15, n. 1, pp. 19-36,1988.

[50] S.S. Gokhale, K.S. Trivedi, “Reliability prediction and sensitivity analysis based on software architecture,”Proc. of the 13th International Symposium on Software Reliability Engineering (ISSRE’02), p. 64, 2002.

[51] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, Brooks/Cole, 1998

[52] S. R. Chidamber and C. F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, 476-493.

[53] V. R. Basili, L. C. Briand, and W. L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering, 22(10), pp. 751-761, 1996.

[54] R. Subramanyam and M. S. Krishnan, Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, IEEE Transactions on Software Engineering, Vol. 29, No. 4 (April 2003) 297-310

[55] R. Subramanyam and M. S. Krishnan, ”Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects”, IEEE Transactions on Software Engineering, 29(4) pp. 297-310, April 2003.

[56] A. B. Binkley, Schach, S., Validation of the coupling dependency metric as a predictor of runtime failures and maintenance measures. In the proc. of the International Conference on Software Engineering, pp. 452 - 455, 1998.

[57] N. Ohlsson, Alberg, H., Predicting fault-prone software modules in telephone switches, IEEE Transactions on Software Engineering, Vol. 22, No. 12, 1996, 886 - 894.

[58] S.S. Gokhale, M.R Lyu, Regression Tree Modeling for the Prediction of Software Quality, in Proc. of the third ISSAT 1997, Anhaiem, CA.

[59] N. Nagappan, T. Ball, A. Zeller, Mining Metrics to Predict Component Failures, In the Proc. of the 28th international conference on Software engineering (ICSE ’06), 2006, 452–461.

[60] G.Denaro, S. Morasca, M.Pezze’, Deriving Models of Software Fault-Proneness, SEKE 2002.

[61] G. Denaro, M.Pezze’, An Empirical Evaluation of Fault-Proneness Models, in proc of the 24th International Conference on Software engineering, 241-251, 2002.

[62] J. C. Munson and T. M. Khoshgoftaax. The detection of fault-prone programs. IEEE Transactions on Software Engineering, 18(5):423-33, May 1992.

[63] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. Using process history to predict software quality. Computer, 31(4):66-72, Apr. 1998.

[64] R.H. Hou, S.Y. Kuo, and Y.P. Chang, “Efficient allocation of testing resources for software module testing based on the hyper-geometric distribution software reliability growth model,” Proc. of the 7th International Symposium on Software Reliability Engineering (ISSRE ’96), pp. 289-298, Oct./Nov. 1996.

[65] Frankl, P.G., Hamlet, D., Littlewood, B., Strigini, L.: Evaluating testing methods by delivered reliability. IEEE Transactions on Software Engineering 24(8), 586-601 (1998)

[66] Y. Nakagawa and S. Miyazaki, "Surrogate constraints algorithm for reliability optimization problems with two constraints," IEEE Trans. Reliability, vol. R-30, pp. 175-181, 1981.

[67] L. Painton and J. Campbell, "Genetic algorithms in optimization of system reliability," IEEE Trans. on Reliability, vol. 44, pp. 172-178, 1995.

[68] F. A. Tillman, C. L. Hwang, and W. Kuo, "Determining component reliability and redundancy for optimum system reliability," IEEE Trans. on Reliability, vol. R-26, pp. 162-165, 1977.

[69] A.O.C. Elegbede, C.Chu, K.H. Adjallah, and F.Yalaoui "Reliability Allocation Through Cost Minimization," IEEE Trans. on Reliability, vol. 52, no. 1, Mar 2003.

[70] N. Wattanapongsakorn and S. P. Levitan, "Reliability optimization models for embedded systems with multiple applications," IEEE Trans. on Reliability, vol. 53, no. 3, Sep 2004.

[71] J. Onishi, S. Kimura, R.J.W. James, and Y. Nakagawa, "Solving the Redundancy Allocation Problem With a Mix of Components Using the Improved Surrogate Constraint Method," IEEE Trans. on Reliability, vol. 56, no. 1, Mar 2007.

[72] F. W. Rice, C.R. Cassady and R.T. Wise, "Simplifying the solution of redundancy allocation problems," Proc. of the Annual Reliability & Maintainability Symposium (RAMS '99), pp. 190-194, 1999.

[73] Rani, R.B. Misra, "Economic Allocation of Target Reliability in Modular Software Systems," Proc. of the Annual Reliability & Maintainability Symposium (RAMS '05), pp. 428- 432, 2005.

[74] A.Mettas, "Reliability Allocation and optimization for complex systems," Proc. of the Annual Reliability and Maintainability Symposium (RAMS '00), pp. 216-221, 2000.

[75] M.R. Lyu, S. Rangarajan and A.P.A. van Moorsel, "Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development," IEEE Trans. on Reliability, vol. 51, no. 2, June 2002.

[76] M. R. Lyu, S. Rangarajan, A.P.A. van Moorsel, "Optimization of Reliability Allocation and Testing Schedule for Software Systems," Proc. of the 8th International Symposium On Software Reliability Engineering (ISSRE '97), pp. 336-347,1997.

[77] Way Kuo and Rui Wan ÒRecent Advances in Optimal Reliability AllocationÓ IEEE Transactions on systems, man, and cybernetics, vol. 37, no. 2, march 2007

[78] M. E.Helander, M.Zhao, N.Ohisson, "Planning Models for software reliability and Cost," IEEE Trans. on Software Engineering, vol. 24, No. 6, June 1998.

[79] W.Everett, "Software Component Reliability Analysis," Proc. of the Symp. Application specific Systems and Software Engineering Technology (ASSET '99), pp. 204-211, 1999.

[80] S. Rapps, E.J. Weyuker, Data ßow analysis techniques for program test data selection. In proc. of the Sixth International Conference on Software Engineering, Tokyo, Japan (September 1982), 272-278

[81] S. Rapps, E.J. Weyuker, Selecting software test data using data ßow information. IEEE Transactions on Software Engineering Vol. SE-14 No.4 (1985), 367-375.

[82] D. Hamlet, Theoretical comparison of testing methods. In proc. of the Third Symposium on Testing, Analysis and Verification, Key West, (1989), 28–37.

[83] J.S. Gourlay, A mathematical framework for the investigation of testing. IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, (1983), 686–709.

[84] E.J. Weyuker, S.N. Weiss, D. Hamlet, Comparison of program testing strategies. In proc. of the Fourth Symposium on Software Testing, Analysis, and Verification, October 1991, ACM Press, New York, 1–10

[85] D. Hamlet, R. Taylor, Partition testing does not inspire confidence. IEEE Transactions on Software Engineering Vol. 16, No.12, (1990), 1402-1411.

[86] P.G. Frankl, E.J. Weyuker, A formal analysis of the fault detecting ability of testing methods. IEEE Transactions on Software Engineering (March 1993), 202-213.

[87] E.J. Weyuker, Comparing the Effectiveness of Testing Techniques, Formal Methods and Testing 2008, 271-291.

[88] J.W. Duran, S.C. Ntafos, An evaluation of random testing. IEEE Transactions on Software Engineering Vol. 10, No. 7, (1984), 438-444.

[89] P. Thevenod-Fosse, H. Waeselynck, Y. Crouzet, An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation. IEEE Fault-Tolerant Computing: The Twenty-First International Symposium, Montreal, Canada, June 1991, 410-417.

[90] P.G. Frankl, S.N. Weiss, C. Hu, All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness, Journal of Systems and Software Vol. 38, No. 3 (1997), 235-253.

[91] M.R. Girgis, M.R. Woodward, An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. In proc. of the IEEE Workshop on Software Testing, July 1986, 64-73.

[92] M. Grindal, B. Lindstrom, J. Offutt, S.F. Andler, An Evaluation of Combination Testing Strategies. Empirical Software Engineering Vol. 11, No. 4 (2006), 583-611.

[93] A. Avritzer, A., E.J. Weyuker, Deriving workloads for performance testing. Software Practice and Experience Vol. 26, No. 6, (1996), 613-633.

[94] A. Avritzer, E.J. Weyuker, Metrics to assess the likelihood of project success based on architecture reviews. Empirical Software Engineering Journal Vol. 4, No. 3, (1999), 197-213.

[95] P.G., Weiss, S.N.: An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. IEEE Transactions on Software Engineer- ing 19(8), 774-787 (1993)

[96] N. Juristo, a. Moreno, S.Vegas, Reviewing 25 Years of Testing Technique Experiments, *Empirical Software Engineering*, 9, 7-44, 2004.

[97] Andras Pfening, Sachin Garg, Antonio Puliafito, Miklos Telek, and Kishor S. Trivedi. Optimal Software Rejuvenation for Tolerating Soft Failures. Performance Evauation, 27-28(4):491-506, 1996.

[98] S. Garg, A. Puliafito, M. Telek, and K.S. Trivedi. Analysis of software rejuvenation using markov regenerative stochastic petri nets. In 6th International Symposium on Software Reliability Engineering (ISSRE 1995), pages 24-27, 1995.

[99] Dazhi Wang, Wei Xie, and Kishor S. Trivedi. Performability analysis of clustered systems with rejuvenation under varying workload. Performance Evaluation, 64(3):247-265, 2007.

[100] Sachin Garg, Antonio Puliafito, Miklos Telek, and Kishor S. Trivedi. Analysis of Preventive Maintenance in Transactions Based Software Systems. IEEE Transactions on Computers, 47(1):96-107, 1998.

[101] Yujuan Bao, Xiaobai Sun, and Kishor S. Trivedi. A workload-based analysis of software aging, and rejuvenation. IEEE Transactions on Reliability, 54(3):541-548, 2005.

[102] S. Garg, Y. Huang, C.M.R. Kintala, and K.S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In ACM SIGMET- RICS Conference on Measurement and Modeling of Computer Systems, Philadelphia,PA (USA), pages 252-261, 1996.

[103] R. Pietrantuono, S. Russo, K.S. Trivedi. Software Reliability and Testing Time Allocation: An Architecture-Based Approach. To appera in *IEEE Transactions on Software Engineering, Special Issue on Software Dependability*, January/February 2010.

[104] S. Yacoub, B. Cukic, and H.H. Ammar,"A Scenario-Based Reliability Analysis Approach for Component-Based Software," IEEE Trans. on Reliability, vol. 53, no. 4, Dec. 2004.

[105] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K.S. Trivedi. A methodology for detection and estimation of software aging. page 283, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[106] K. Vaidyanathan and K.S. Trivedi. A measurement-based model for estimation of resource exhaustion in operational software systems. pages 84-93, 1999.

[107] Karen J. Cassidy, Kenny C. Gross, and Amir Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In 2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings, pages 478-482. IEEE Computer Society, 2002.

[108] Lei Li, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. An approach for estimation of software aging in a web server. In 2002 International Symposium on Empirical Software Engineering (ISESE 2002), 3-4 October 2002, Nara, Japan, pages 91-102, 2002.

[109] M. Grottke, L. Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of software aging in a web server. IEEE Transactions on Reliability, 55(3):411 - 420, Sept. 2006

[110] L. Silva, H. Madeira, and J.G. Silva. Software aging and rejuvenation in a soap- based server. In Proc. of 5th International Symposium on Network Computing and Applications (NCA06), Cambridge,MA (USA), pages 56 - 65, 2006.

[111] Gunther A. Hoffmann, Kishor S. Trivedi, and Miroslaw Malek. A best practice guide to resources forecasting for the apache webserver. In 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA, pages 183-193. IEEE Computer Society, 2006.

[112] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, An experiment in memory leak analysis with a mission-critical middleware for Air Traffic Control, in proc. of the 1st Workshop on Software Aging and Rejuvenation.

[113] A. Pasquini, A. N. Crespo, and P. Matrella, "Sensitivity of reliability-growth models to operational profile errors vs testing accuracy," IEEE Trans. on Reliability, vol. 45, no. 4, pp. 531-540, 1996.

[114] V.S.Sharma, K.S.Trivedi, "Quantifying software performance, reliability and security: An architecture-based approach," The Journal of Systems and Software, vol. 80, Issue 4. pp. 493-509, April 2007.

[115] V. Almering, M. Van Genuchten,G. Cloudt, P.J.M. Sonnemans, "Using Software Reliability Growth Models in Practice," IEEE Software, vol. 24, no. 6, pp. 82-88, Nov.-Dec. 2007.