UNIVERSITÀ DEGLI STUDI DI PISA

DIPARTIMENTO DI INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA

PH.D. THESIS

Software Testing in the XXI Century: Methods, Tools and New Approaches To Manage, Control and Evaluate This Critical Phase

Eda Marchetti

THESIS COMMITTEE
Antonia Bertolino (Supervisor)
Ugo Montanari
Carlo Montangero

REVIEWERS
Lionel C. Briand
Shari Lawrence Pfleeger

September 30, 2003

To my son Matteo

Acknowledgments

I wish to thank all the people who in different way have contributed to the realization of this Thesis. Surely I would never have arrived at this result without their support, stimuli, suggestions, patience and love.

I thank Antonia Bertolino who introduced me to the stimulating environment of Software Testing. In these years she constantly offered me her professional and personal support and was for me not only a good teacher but also a dear friend. Within her research group I found a stimulating, inspiring and friendly environment and had the good fortune to collaborate with invaluable people such as: Francesca Basanieri, Francesca Martelli, Andrea Polini, Alberto Ribolini, and everyone else in the Pisatel Lab.

Likewise, I wish to thank Raffaela Mirandola for her productive collaboration and the precious advice and encouragement, and Gaetano Lombardi, Giovanni Nucera and Emilia Peciola for the important research stimuli and useful discussions.

I thank my husband Massimo for his love, understanding, and support throughout these years and without whom I probably would have never reached this result. And a special thanks to my little baby Matteo who awaited me patiently at home during the writing of this Thesis.

I am also grateful to my parents and my sister Sara for their love, understanding and continuous support in all the difficulties encountered during these years.

Thanks finally to all my friends who have been close to me and have tolerated my moods and strangeness.

Contents

1 Introduction

In the twenty first Century, it seems impossible to think that less than 50 years ago people could live without software applications. Nowadays we have the impression that everything we use, every “electronic contraption” in our house, office or car contains a software (firmware) heart, but so is. From their first few months, children are accustomed using interactive or musical toys, playing videogames, listening to music or interacting with computers. Our kitchens are filled with domestic appliances; our cars are more like computers with four wheels than mechanical devices; very rarely does an office exist without at least one personal computer; even for writing this Thesis we are using a software application (indeed writing it without this support was unthinkable). However, these are just trivial examples; software systems are vastly applied in every industrial and medical field, they control air, sea, and road traffic and often are responsible of our lives.

Since the 1980, the widespread use of these technologies has led a large part of the software engineering to focus its attention on quality, usability, safety and other characteristic attributes of software applications. In particular, interest was captured both by the process for software development and by its results. Using their experience software engineering researchers have gradually arrived at the conviction that only the joint between a mature and well-established development process with specific techniques for the quantitative evaluation of the attributes of interest of the artefacts produced can guarantee high quality and reliable applications. Therefore research has been split into two sets, with of course some natural intersections and points of contact: the former interested to the process (Software Process Improvement (SPI) [KK00]), and the latter focused on the product.

Frameworks such as CMM [PPC93], SPICE [DO99], RUP [RUP] (detailed in Chapter 3) are the “products” of the SPI research work belonging to the former set. They capture the good practices for the process assessment and are de facto references used by thousands of organizations.

Considering the latter set, generally the techniques applicable to the product can be divided into two groups: static techniques, which do not involve code execution,

as such for instance inspection, reviews, code reading, and the dynamic techniques, which instead require code running, to which testing belongs (Chapter 2). Among these diverse techniques in this Thesis we concentrate on Software Testing, which is a means applicable for both evaluating product quality and improving it indirectly, by identifying defects and problems. We propose a global view of the testing phase exploiting and unifying the knowledge both from the industrial reality and research context. In particular we will go offer the readers methods, tools and new approaches, each evaluated in terms of effectiveness, cost and applicability by means of case studies derived also from real industrial contexts, useful for planning, monitoring, and controlling the different stages of testing process.

Software Testing concerns many related activities developed in view of specific purposes (Chapter 2). In particular, by the application of well-defined techniques, it exposes software failures that may involve the whole system, parts of it, or even a single module. The failures are the primary object of interest during testing activities and they are evaluated by measurement process for obtaining values of interest concerning the program under test.

During recent years Software Testing has increased its role in the process of development. It is no longer focused on the defects detection after code completion, but is now an integrated and significant activity performed during the whole software life cycle. Its critical nature and the importance for the overall quality of the final products led adopting the good practice of starting its management at the early stages of software development during the requirements analysis and proceeding with its organization systematically and continuously during the entire development process up to the code level.

Of course, the management of the testing activities depends strictly on the development process adopted for delivering the software products; however the main phases can be resumed in [BE01]:

Planning: As for any other process activity, the testing must be planned and scheduled. Thus the time and effort needed for performing and completing Software Testing must be established in advance during the early stages of development. This also includes the specification of the personnel involved, the tasks they must to perform and the facilities and equipments they may use.

Test cases generation: According to the test plan constraints a set(s) of the test cases must be generated by using a (several) test strategy (ies).

Test cases execution: The test cases execution may involve testing engineers, outside personnel or even customers. It is important to document every action performed in order to allow the experiments' duplication and meaningful and truthful evaluations of the results obtained.

Test results analysis: The collected testing results must be evaluated to determine whether the test was successful (the system performs as expected, or there are no major unexpected outcomes) and used for deriving measures and values of interest.

Problem reporting: A test log documents the testing activity performed. This should contain for example the date in which a test was conducted, the data of the people who performed the test, the information about the system configuration and any other relevant data. Anomalies or unexpected behaviours should be also reported.

Post-closure activities: the information relative to failures or defects discovered during testing execution are used for evaluating the performance and the effectiveness of the developed testing strategy(ies) and determining whether the process development adopted needs some improvements.

In this Thesis, considering the above subdivision of activities, we examine several difficulties concerning the applicability of Software Testing in the industrial contest. In particular, starting from the test planning we proceed systematically with the analysis of the different testing stages, pointing out the characteristic problems and presenting our original proposal for solving them.

1.1 Motivations and Objectives

Software Testing is a critical part of the process of development, on which the quality of the products delivered strictly depends. Testing activity, as reflected in the above phases' subdivision, is in fact not limited to the detection of software "bugs" but it encompasses the whole development process. Specifically, it has been evaluated that testing consumes at least half of the labour (calculate in terms of required time/effort/resources/people) expended producing a deliverable software product and sometimes, in the case of critical systems, may even reach 90% [BE90]. Thus Software Testing must be well-planned and executed, otherwise severe consequences can result. Recent reports and industrial experiences are testimonies that an erroneous evaluation of the product's quality, which allows the release of products with important residual defects, may have negative consequences with huge

loss of revenue and also risking the users' safety. Unfortunately many times due to time or cost constraints Software Testing is not developed in the proper manner or is even skipped.

The approaches and methods presented in this Thesis share a general aim: put research in practice. Our methods are also the fruits of many constructive discussions with project managers, testers and developers, who bring up real necessities not yet satisfied thoroughly by the researches performed so far.

Therefore for defining our proposals the procedural steps followed were first going over the literature in depth, studying and evaluating the proposed solutions relative to test planning, test cases derivation and test results analysis. Then for facing industries' needs, either readapting and improving the approaches found in the literature or defining original alternatives solutions, finally proposing systematic and rigorous procedural methods.

For this we paid particular attention to strategies for selecting the parts (functionalities) of the software products on which the testing must concentrate in order to avoid loss of time and effort. In the literature several solutions are presented for generating suitable test cases, but the authors seldom concentrated on the methodological approaches for the selection of the functionalities to be tested. Generally this is a crucial aspect for software developers, which is often left to the intuition or expertise of the program managers or testers. Unfortunately wrong decisions in this contest can considerably increase the overall effort and time required for delivering a "good" product. Thus in this Thesis we specifically concentrate on this problem by proposing procedural strategies, which guide us to suitable testing choices from the first phases of process development.

During the methodology development, following the principle of finding applicable solutions for the industrial environment, we were subjected to two important constraints: maximizing the usability and automation.

Concerning usability, software developers want easy-to-use and ready-to-apply methodologies, which minimize as much as possible the required additional formalism or ad-hoc effort specific for testing purposes. In the industrial context these aspects are immediately translated into an increase in the cost of Software Testing, which is improbably justified and accepted even with the evidence of extremely good results and a great improvement in software quality. Therefore our objective was presenting systematic and rigorous methodologies, which as far as possible will adapt themselves to modelling notations and procedures commonly

used by industries and real environments and not vice versa. Of course achieving the optimal trade-off between usability and high-quality methodology is a difficult task, which may sometimes interfere with the improvements of the solutions proposed.

The second constraint considered is automation. The increasingly strict delivery time imposed by the customers and markets forces software developers to accelerate product development as much as possible. This often is translated into reducing the time necessary for performing Software Testing, which is one of the most expensive activities of the development. Consequently the testing phase is partially skipped and the software products released in advance only because there is not enough time for testing them properly.

One of the ways to pull down the overall testing time is to considerably increase if possible the automation in test cases derivation, execution and validation, thus reducing the manual labour. Considering these problems in this Thesis we adopt automation as a leading principle for our proposals. Therefore, for each of them we present executable prototypes or we define the potential architecture completely, implementing only some of the involved components. In particular, in this process scrupulous attention has been dedicated to the automatic selection of functionalities to be tested and the consequent derivation of test cases, which allowed us to considerably reduce the time required for the Test Plan definition.

As a side effect the collaboration with the industrial world provides us with interesting case studies used for evaluating the effectiveness of the proposals of this Thesis. The results obtained highlighted the peculiarities and deficiencies of our methodologies and were the stimuli for modifications or improvements in order to better adapt them to the software developers' demands.

Thus in this Thesis we not only provide theoretical advancement, but also focus specifically on the definition of practical and quantitative support applicable all along the testing phase. As will be further detailed in the next section, our contribution concerns: providing an original approach for scheduling the testing activities and distributing people and resources among them considering a multiproject environment; defining a tool which supports the user both in the choice of the most important software elements on which the testing effort must be concentrated, and in the automatic generation of the appropriate test cases by using the available UML product specification; evaluating the effectiveness of the testing techniques applied, while the tests are executed, in order to decide when stop testing. In particular, we present for each of the topics treated a detailed literature survey and a quantitative

analysis of the methodologies proposed by means of their application to case studies. Moreover, the comparison with other alternative solutions taken from the literature is also provided.

1.2 Thesis Outline

In this Thesis we overview the different stages of the testing process from the planning to the effective run of test cases and evaluation of the obtained results considering division of activities previously presented for organizing the contents. In particular we have divided the Thesis into five self-contained parts, each related to a different testing stage, excluding the first which is an introductory section. Therefore, these reflect the life cycle of software and are in a temporal relation each other, even if several intersections and contact points exist.

Part I: The first part presents a overview of Software Testing and provides a brief description of the Unified Modelling Language and the Rational Unified Process that will be used in the Thesis. This part aims to give readers the basic information they need for a complete comprehension of the methods and approaches presented. Specifically:

Chapter 2

We provide a comprehensive view of the Software Testing, clarifying the terminology that will be used and bringing the relevant issues together in a unified context.

Chapter 3

We provide here a brief description of the Unified Modelling Language and the Rational Unified Process extensively used in Chapter 4, 5 and 8.

Part II: In *Chapter 4* we begin our journey into Software Testing considering the planning activity. This must start from the early stages of requirement analysis and continue with further refinements during the entire development. Indeed establishing a suitable test plan is not a trivial task because it includes the definition, assignment and scheduling of resources, time, personnel and costs. This task is even more critical in a multiproject environment in which resources and personnel are shared among the various products realization. Thus, judging whether the resources assigned to a specified task are adequate or whether under the existing organizational

schemes the predicted time schedules will be met is a very difficult task also because the processes involved are highly complex. The influencing factors (both human and technical in kind) are in fact many, and in most cases not easily measurable or predictable.

Our response to these problems is Propean, an integrated approach in which managers can define UML models of the flow of activities to be performed during development and the tasks to distribute among personnel by using familiar notations and tools and then derive automatically the measures of interest. Propean is based on the techniques of computer software performance engineering and queueing networks. It adopts the following metaphor: the project teams correspond to the processing resources in performance models. The process activities are associated with the tasks to be accomplished within established time intervals. Propean allows estimations of time necessary for completing the different testing activities, with respect to the established deadline, and the utilization rate of each resource (people).

Contrarily to some existing tools Propean deals with multiproject environments and provides predictions which rely on a solid mathematical background and have statistical validity. Propean automatically translates the models into a format that is processable by standard performance analysis algorithms and applies a solver of the latter to obtain the desired results. In Chapter 4 we show how the well-known techniques from performance analysis can be usefully and quite naturally adapted to tasks of relevance for software managers, such as assessing the time to completion of specified activities, handling personnel multitasking over different projects, optimising the workloads in development cycles, deciding about product release, and similar issues.

Part III. In this part we proceed with our exploration considering a subsequent activity: tests generation. This is one of the most expensive phases of testing development. On the basis of the financial plan established, the test cases must be defined and distributed among the functionalities of the system to be tested, and then executed. But deciding both the functionalities on which the testing effort should be concentrated and the amount of test cases to dedicate to each of them is another critical point for the testing management. Wrong decisions could increase considerably the overall cost of the testing phase and the time required for its completion.

In *Chapter 5* we propose an integrated, practical and automatic approach, called Cow_Suite, which is also prototyped in a tool for generating and planning a suitable set of test cases, starting with the UML documentation. This methodology combines two original components working in agreement: a strategy called Cowtest and a method called UIT. The former provides two different test planning schemes: testing must respect a certain resource investment, which in practice we translate into fixing the number of test cases; or the test cases must cover a fixed percentage of functionalities. The latter automatically generates test suites for the high-level test stages, encompassing system and integration testing at various levels. Each generated test suite focuses on a functional portion of the system as interactively selected by the tester on a suitable structure of the UML diagrams.

Cow_Suite has diverse and important characteristics that can be summarised as: usability, using exactly the same UML diagrams developed for analysis and design for test planning without requiring additional formalism; timeliness, starting test generation and selection as early as possible in the development cycle (even from analysis or design phases); incrementality, considering progressively larger parts of the system and addressing, at each incremental step, the functionalities and interactions that are relevant at the level considered; scale, ability to manage even big test suites keeping their sizes and functional coverage under control.

Part IV: In this part we consider the final stages of testing: the analysis of testing results. Despite the effectiveness of testing techniques applied, obtaining a defect-free code remains wishful thinking. Coping with software failures, during development and after release, is among the hardest tasks of managers, while testing, debugging and maintenance activities still consume the largest part of development effort and resources. Each failure requires meticulous extra work in order to find the causing fault(s) and correct it, which could contribute to an expected enormous increase in the final cost of the testing phase. Thus methods to estimate the defect contents of software are of great interest for managers and testers.

Thus we focus our attention on the methods for predicting the final failure rate using the test results obtained during the testing execution. In particular, we consider two different situations: the results are relative to the running of non-operational testing or the failures collected are from operational testing execution. Specifically:

Chapter 6

We propose two dynamic methodologies, the One-Step and the Two-Steps Method for deriving the number of failures experienced up to the end of testing phase, by using data collected during the testing itself. Having this estimation in advance allows the software developers to suddenly take corrective action and drastically reduce the extra cost of the testing phase.

The most attractive feature of the proposed models is their simplicity: they only need collecting the intervals of time between subsequent failures without requiring estimation of parameters of the product or of the development process. Specifically, for prediction purposes in both the One-Step and the Two-Steps Method we used a Classical estimator and an alternative Bayesian estimator.

Even if the models are conceived for dealing with non-operational test results, their generality also allows their application to operational failure data as will be described in this Chapter.

Chapter 7

In this Chapter we continue the exploration of the testing phase considering operational testing. The data obtained during this phase are generally used for the application of reliability growth models, which let the evaluation of some product characteristics such as the level of reliability attained. Unfortunately there is currently no known method for determining a priori which model will prove optimal for a particular development effort. Thus an important role in facilitating the reliability growth model selection and usage is due to the available tools. For this we describe the necessary steps for using two of these tools: SoRel and CASRE. We discuss their respective roles, the former in verifying the basilar assumption that the failure data exhibits a growth in reliability, the latter in selecting the suitable reliability growth model for obtaining the required prediction. We then discuss the advantages and difficulties encountered in applying these models for reliability prediction, and also describe a procedure describing the steps necessary for the integrate use of SoRel and CASRE.

Part V: In *Chapter 8* we present the conclusions and an ongoing experience concerning the readapting and integration of the Cow_Suite methodology into a more general framework for enabling the validation of Component Based (CB) systems by testing them against the corresponding UML architectural specifications. Specifically

using the emerging methodologies for representing components and development process [CD00], we outline the steps necessary to provide the user with a tool for components testing. Our intention is to define a test environment, called UML Combination, which will be the joint, with the necessary adaptations, of Cow_Suite, for analyzing the UML components specification and selecting and generating test cases, and CDT [BP03] for codifying the test cases and (re-)executing them every time a component instance will be plugged into the system.

1.3 An overview of the publications status

The research proposals of this Thesis have been obtained with join collaboration of different people both from the academic and industrial world and several results achieved have been published in different international conferences proceeding and journals. Specifically here for each Chapter we highlight those sections have already been published and those not yet.

Chapter 4

We presented the Propean methodology in [BMM02], in which both the details of the methodology and the architecture of the Propean Tool are described. The application of Propean to managing the testing phase and its use with RUP are instead published in [BMM02] and [BLM02] respectively. Propean has been applied to further a case study not included in this Thesis and presented in [BBM02a] and [BBM03].

Chapter 5

The description of the test strategy Cowtest and the methodology Cow_Suite have been presented in [BBM1] and [BBM02] respectively. In this Chapter we improve the overview of literature and include more details about the steps necessary for Cow_Suite application, the procedure adopted for the derivation of the basic structure and the algorithms used for the test cases derivation. The comparison between the test plan derived applying Cow_Suite and that produced manually by the ERI personnel has been published in [BIL03].

Chapter 6

The description of the One-Step and Two-Steps methods and their application to the real cases study have been published in [LPM99] and [BMM02a], while the Bemar model has been presented in [BM01].

Chapter 7

The application of the Reliability Growth Models to industrials cases study has been published in [BLM98]. Here we also present an original procedure for the integrated use of the tools Sorel and CASRE with the purpose of reliability prediction.

Chapter 8

The ongoing extension of Cow_Suite to the Component based paradigm has been published in [BMP03], here we include more details of the combined application of the existing tools.

The following table summarizes the Thesis proposals, presenting our research result for each testing phase.

Testing phase	Research Results
Planning	Propean approach for Test Planning Management using Queueing Networks (Chapter 4) [BBM02a, BLM02, BMM02, BBM03]
Test cases generation	In OO environment Cow_Suite for test case generation and selection (Chapter 5) [BBM01, BBM02, BIL03]
	In Component Based environment a readapting of Cow_Suite for the "UML Components" designs (Chapter 8) [BMP03]
Test results analysis	Non-operational testing: One-Steps and Two-Steps methodology for predicting the cumulative number of failures (Chapter 6) [BM98, LPM99, BMM02a]
	Operational testing: application of the Reliability Growth model for reliability predictions (Chapter 7) [BLM98]

Table 1 The research proposals in the different testing phases.

PART 1:
SOFTWARE TESTING: BASIC KNOWLEDGE

2 Software Testing

Preface

This Chapter attempts to provide a comprehensive view of Software Testing, clarifying the terminology that will be used in this Thesis and emphasizing the relevant issues in a unified context. Due to the vastness of this topic, for each subject we only provide a brief description and set of guideline references. We defer to the different Chapters of this Thesis for a more complete and exhaustive explanation.

In particular, in Section 2.1 we present the definition of testing as well as some alternative (static) techniques that can be applied for software quality purposes, while in Section 2.2 we differentiate the various testing stages applicable during the testing process. The techniques relevant for selecting the proper set of test cases are depicted in Section 2.3, while the methodologies for evaluating the testing results are found in Section 2.5.

2.1 Testing Phase

The testing phase is an important and critical part of software development, consuming even more than half of the effort required for producing deliverable software [BE90]. Unfortunately, often due to time or cost constraints, the testing is not developed in the proper manner or is even skipped. The testing activity in fact is not limited to the detection of “bugs” in the software, but it encompasses the entire development process. The testing planning starts during the early stages of requirement analysis, and proceeds systematically, with continuous refinements during the course of development until the completion of the coding phase, with the beginning of the test cases execution. This last step represents the biggest part of software cost that can be evaluated in terms of: the cost of designing a suitable set of test cases which can reveal the presence of bugs; the cost of running those tests, which also requires a considerable amount of time; the cost of detecting them, i.e. the development of a proper “oracle” which can identify the manifestation of a bug as soon as possible; the cost of correcting them.

All these activities have in common the same testing purpose: evaluating the product quality for increasing the software engineering confidence in the proper functioning of the software. However it must be made clear that testing cannot show the absence of defects; it can only reveal that software defects are present, as shown by Dijkstra as long as thirty years ago [DI70].

We report the definition of the Software Testing introduced in [BE01]:

Software Testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behaviour

As shown by this definition, testing deals with dynamic verification of system quality, which also involves the code execution, as will be better described in this Chapter.

Generally the techniques applicable for quality evaluation can be divided into two sets: static techniques, which do not involve code execution, and dynamic techniques, to which testing belongs, which instead required running code. The static techniques are applicable all during the process development for different purposes such as to check the adherence of the code to the specification or to detect defects in code by its inspection or review. Instead the latter approach more properly observes failures as they show up. In particular dynamic analysis techniques involve the execution of the code and the analysis of its responses in order to determine its validity and to detect errors. The behavioral properties of the program are also observed.

Other examples of dynamic analysis include simulation, sizing and timing analysis, and prototyping, which may be applied throughout the lifecycle [PW93]. In this Chapter we briefly present the static techniques (Section 2.1.1), preferring to concentrate on testing, which is the main topic of this Thesis[1].

Before continuing the presentation of the diverse aspects of Software Testing it is important to clarify the terminology relative to the terms “fault”, “defect” and “failure” that we will use. Although their meanings are strictly related, there are some distinctions between them.

As discussed in [BE03], a *failure* is the manifested inability of the program to perform the function required, i.e. a system malfunction evidenced by incorrect output, abnormal termination or unmet time and space constraints. The cause of a

[1] For the structure of the following sections we refer to [BE01].

failure, i.e. the missing or incorrect code, is a *fault*. In particular, a fault may remain undetected until some stirring up event activates it. In this case it brings the program into an intermediate unstable state, called *error*, which if propagated to the output causes a failure. The process of failure manifestation is therefore [PL98]

$$\text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure}$$

which can be iterated recursively: a fault can be caused by a failure in some other interacting system.

Testing reveals failures and a consequent analysis stage is needed to identify the faults that caused them. In particular, it is possible that many different failures can result from a single fault, and the same failure can be caused by different faults. In this situation what should be better estimated in a program, its number of contained "faults" or how many "failures" it would expose? Either estimate taken alone can be tricky: if failures are counted it is possible to end up the testing with a pessimistic estimate of program "integrity", as one fault may produce multiple failures. On the other hand, if the faults are taken into consideration, it is possible to evaluate at the same level harmful faults that produce frequent failures, and inoffensive faults that may remain hidden for years of operation. It is hence clear that the two estimates are both important during development and are produced by different (complementary) types of analysis.

In this Thesis we will present different methodologies and approaches to manage and control the testing phase under different complementary aspects. In particular, we are interested in what it is observable by testing, i.e., the failures.

2.1.1 Static Techniques

The static techniques are based on the examination of the project documentation, the software and other related information about requirements and design and not on software execution [DRW02]. These data are also used to trace the requirements into the developed software and to verify its adequacy to the specification. The static techniques include software inspection, reviews, code reading, algorithm analysis and tracing. Thus the use of static techniques in not limited to the testing phase; their application during the entire the development phase is even more important. In particular they can be applied [PW93]:

- During the requirements phase, they can be used to check adherence to specification conventions consistency, completeness and language syntax.

Commonly used static analysis techniques are flow analysis, data flow analysis, traceability analysis, and interface analysis.

- During the design phase, the most commonly used techniques include algorithm analysis, database analysis, interface analysis and traceability analysis.
- During the implementation phase the frequently used techniques are code reading, inspections, walkthroughs, reviews, control flow analysis database analysis interface analysis, and traceability analysis. Other categories of techniques are complexity analysis, sneak circuit analysis and slicing.

It is worth noting that an important and widespread used class of static (analytic) techniques is the use of formal methods to verify software requirement and design. This technique is attracting quite a lot attention from both research and industries and the proof of correctness as well as the verification of security and safety requirements of different (crucial) parts of a critical system is will be increasingly applied [NAS97]. However this is a vast area of research, quite far from the objectives of this Thesis, thus we refer the reader to [WR01] for complete documentation and overview of the literature.

Nevertheless, it should be considered that a defect can be more or less disturbing depending on whether, and how frequently, it will eventually show up at the final user (and depending of course on the seriousness of its consequences). Indeed, whether few or many, some defects will inevitably escape testing and debugging. So, in the end, one important measure of quality of the product useful in deciding whether it is ready for release is software reliability. Until they do not cause failures the remaining defects trouble neither customers nor producers. In Chapter 7 we will discuss this topic in detail.

We conclude this section considering the alternative application of static techniques in producing values of interest for testing process control. Different estimations can be obtained by observing specific properties of the present or past products, and/or parameters of the development process In particular during the testing phase the static techniques may be applied to estimate the total number of defects and provide very attractive measures. Since by testing we find the failure and fix the related fault, static models would provide a prediction on how many defects are left in the code.

Thus static techniques could be very attractive to managers for prediction purposes, because they provide "numbers", which the managers are eager for, very early in the process compared to dynamic models. The latter can be used late in the

life cycle, i.e., during the testing phases, when it may be too late to efficaciously re-direct development efforts. Static defect models on the other hand can be applied to identify more risky modules and consequently re-allocate testing resources or modify design.

It is not possible to decide which is the most appropriate between static and dynamic techniques because both of them are useful for different objectives. For instance in the front-end phases of the life cycle, managers should use the static to apportion risk among modules and to allocate development time and resources. In the final stages of development they should use the dynamic instead in order to evaluate the degree of disturbance created by defects that are inevitably left, and to decide whether the product is ready for delivery.

2.1.2 Definition of Software Testing

Referring to [BE01, BE03] in this section we discuss the main concepts of the Software Testing definition provided in Section 2.1. As already stated, the testing is a complementary approach of the static techniques described which involves the execution of the code. The term "dynamic" means precisely that the *"testing always implies executing the program on (valued) inputs"* [BE01] in a specific environment. Principally for the non-deterministic systems, the results obtained by testing depend strictly on the input provided as well as state of the system. Therefore when speaking about input values the definition of the parameters and environmental conditions characteristic of a specific system state must be included when necessary.

Of course, even if the set of input values can be considered infinite, those that will be run effectively during the testing of a program must be finite. It is in practice impossible, due to time constraints, to exhaustively exercise every input of a specific set even when not infinite: this operation could require thousand of years [DJ70]. As stated in [BE03], a good test strategy therefore requires a trade-off between the number of chosen inputs and overall time and effort dedicated to the testing purposes. The selection of test cases is thus a critical and important aspect of testing, and the choice of the best *test criterion* to be applied for this purpose is a complex problem as yet unsolved [VJB03]. Different techniques can be applied depending on the target and the effect that should be reached.

Once the tests are selected and run, another crucial aspect of this phase is the detection of failure, i.e. *the oracle problem*, which means deciding whether the observed outcomes are acceptable or not. As reported in [BE01] there are two

possibilities for checking the behavior of the program under testing: *testing for validation,* i.e., evaluating the program against the user's expectations, or *testing for verification* (conformance testing), i.e., evaluate the program against the specifications.

2.2 Testing Level

Generally the testing is performed at different levels during the development process and can involve the whole system or parts of it. Here we distinguish three different stages: unit, integration and system test [BE90 Chapter 1], [PL98 Chapter 7] providing in the following a brief description of each. It is important to clarify that no stage is more important than another. Each one has its specific target and difficulties and only a good combination of them can provide products of quality.

Unit Test

A unit is the smallest testable piece of software, consisting of hundreds or a few lines of source code, and generally representing the result of the work of one programmer. The Unit test's purpose is to ensure that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure [BE90, PL98]. When the tests reveal an anomalous behavior, it is said that there is a unit bug.

Unit tests can also be applied for test interfaces (parameter passed in correct order, number of parameters equal to number of arguments, parameter and argument match), local data structure (improper typing, incorrect variable name, inconsistent data type) or boundary condition. Further specific details concerning the unit test are in [IEEE93].

Integration Test

Integration is a process by which components are aggregated to create a larger component. Even though the single components are individually acceptable when tested in isolation, they could result incorrect or inconsistent when combined in order to build complex systems. For example, there could be an improper call or return sequence between two or more components [BE90]. Therefore integration testing is specifically aimed at exposing the problems that arise from the combination of components by the verification that each component behaves according to its specification defined during preliminary design. In particular, it is mainly focused on

the communication interfaces among integrated components. The recent (and even not-so-recent) testing literature contains few entries relative to integration testing, and practical methodologies rely essentially on good design sense and the testers' intuition.

Integration testing of traditional systems was done substantially in either a non-incremental or an incremental approach. Except for small, simple systems, in a non-incremental approach the components are linked together and tested all at once (big-bang testing) [JO95]. In the incremental approach, we find the classical "top-down" strategy, in which the modules are integrated one at a time, from the main program down to the subordinated ones, or "bottom-up", in which the tests are constructed starting from the modules at the lowest hierarchical level and then are progressively linked together upwards, to construct the whole system. Usually in practice, a mixed approach is applied, as determined by external project factors (e.g., availability of modules, release policy, availability of testers and so on) [PL98].

In modern Object Oriented, distributed systems, approaches such as top-down or bottom-up integration and their practical derivatives, are no longer usable, as no "classical" hierarchy between components can be generally identified. Some other criteria for integration testing implies integrating the software components or subsystem based on identified functional threads [MGB99], [MU02]. In this case the test is focused on those classes used in reply to a particular input or system event (thread-based testing) [Jo95]; or by testing together those classes that contribute to a particular use of the system.

A different branch of the literature is testing based on the Software Architecture: this specifies the high level, formal specification of a system structure in components and their connectors, as well as the system dynamics. Some recent papers explore the way in which the description of the Software Architecture could be used to drive the integration test plan [BCI00, BIM01, MU02]. Indeed, in [GKC01] the authors have also investigated the expression of Software Architecture in UML, with appropriate stereotype extensions.

Finally, some authors have used the dependency structure between classes as a reference structure for guiding integration testing, i.e., their static dependencies [KGH95], or even the dynamic relations of inheritance and polymorphism [LTW00]. Such proposals are interesting when the number of classes is not too big; however, test planning in those approaches can begin only at a mature stage of design, when the classes and their relationships are already stable.

In this Thesis (specifically in Chapter 5) we concentrate mainly on this typology of testing, proposing in contrast an approach that deals with big, complex systems and that can be used from the early stages of design, when the component structure is preliminarily sketched.

System Test

System test involves the whole system embedded in its actual hardware environment and is mainly aimed to verify that the system behaves according to the requirements document. In particular it attempt to reveal bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects.

As will be discussed in Chapter 7, test and data collected applying this type of testing can be used for defining an operational profile of the system which support a statistical analysis of the systems reliability [MU93], [LY96]. Generally system testing includes testing for performance, security, reliability, stress testing and recovery [JO95, PL98].

2.2.1 Objectives of Testing

The Software Testing can be applied for different purposes, such as verifying that the functional specifications are implemented correctly or that the system shows specific reliability. In [BE01] a complete list of the possible testing objectives is provided; here we limit ourselves to describing those that will be mentioned in this Thesis.

- **Acceptance testing** is the final test action prior to deploying the software. Its goal is to verify that the software respects the customer's requirement, i.e., it can be used by the end-users to perform those functions and tasks the software was built to do [PL98].
- **Alpha testing** Before releasing the system it is given to the in-house user for exploring the functions and business tasks. Generally there is no test plan to follow; the individual tester determines what to do [KFN99].
- **Beta Testing** the same as Alpha testing but the system is given to external users. In this case the amount of detail, the data, and approach taken is entirely up to the individual tester. Each tester is responsible for creating their own environment, selecting their data, and determining what functions, features, or tasks to explore. Each tester is responsible for identifying their own criteria for whether to accept

the system in its current state or not. Beta testing is thus the less controlled phase [KFN99].

- **Reliability achievement:** testing is a means to improve reliability; therefore the test case must be randomly generated according to the operational profile. In Chapter 7 we provide a complete description of reliability testing as well as the method used for evaluating the reliability level reached [LY96].
- **Functional Testing**: Tests focused on validating whether the observed behavior of the tested system conforms to the specification. In particular it checks whether the functions are as intended and provides required service(s) and method(s). This test is implemented and executed against different tests targets, including units, integrated units, and systems [PE95]

2.3 Functional and Structural Testing

As stated previously, a testing technique is a systematic method used to select and/or generate tests. It can be considered effective if the tests included are likely to reveal bugs in the tested object. Since objects are modified in order to correct their bugs, the kind of bugs found in an object changes with time, and thus the effectiveness of a technique.

In [BE01] two alternative classifications of test techniques are provided: the first is based on how the tests are generated (for instance tester expertise, specification, code structure and so on) the second is based on the type of information about the software, used for generating the tests (black-box or white-box). We adopt the latter here briefly presenting the main testing techniques applicable.

2.3.1 Functional Testing

Functional testing, also called black box testing, relies on the input/output behaviour of the system. In particular the system is subjected to external inputs, so that the corresponding outputs are used to verify the conformance of the system to the specified behaviour, with no assumptions of what happens in between. Therefore in this process we assume knowledge of the (formal or informal) specification of the system under test, which can be used to define a behavioural model of the system (a transaction *flowgraph).* This graph is either focused on how software is built (i.e. structure) or on how it behaves (i.e. function). A structural focus leads us to structural test techniques, whereas a functional (behavioural) focus leads us to functional test methods [BE90].

One of the crucial aspects of black box testing is therefore the inputs selection. A complete functional test would consist of subjecting the program to all possible input streams and verifying the outcome produced, but as stated in Section 2.1.2 this is theoretically impossible. For this different techniques can be applied such as:

- Testing from formal specifications: In this case it is required that specifications be stated in a formal language, with a precise syntax and semantics. The tests are hence derived automatically from the specification, which are also used for deriving inductive proofs for checking the correct outcome [ZHM97].
- Equivalence partitioning: the functional tests are derived from the specifications written in structured, semiformal language. The input domain is partitioned into equivalence classes so that elements in the same class behave similarly. In this context the Category Partition is a well-known and quite intuitive method, which provides a systematic, formalized approach to partition testing [OB88].
- Boundary-values analysis. This is a complimentary approach to equivalence partitioning, and concentrates on the errors occurring at boundaries of the input domain. The test cases are thus chosen near the extremes of the class. [JO95, KFN99].
- Random methods: consist of generating random test cases based on a uniform distribution over the input domain. It is a low-cost technique because large sets of test patterns can be generated cheaply without requiring any preliminary analysis of software [BE90].
- Operational profile: test cases are produced by a random process meant to produce different test cases with the same probabilities with which they would arise in actual use of the software [LY96].

One of the points against the black-box testing is its dependence on the specification's correctness and the necessity of using a large amount of input in order to get good confidence of acceptable behaviour.

2.3.2 Structural Testing

The structural testing, also called white-box testing, requires complete access to the object's structure and internal data, which means the visibility of the source code. The tests are derived from the program's structure, which is also used to track which parts of the code have been executed during testing. For this some of the commonly used techniques for test case selection are:

- Control flow-based criteria: these techniques use the control flow graph representation of a program in which nodes correspond to sequentially executed statements while edges represent the flow of control between statements. The aim of white box testing criteria is to cover as much as possible the control flow graph, limiting the number of selected test cases. In particular they differentiate in: *statement coverage* which is based on executable statements, *Branch coverage*, which focuses on the blocks and case statements that affect the control flow, *Condition coverage* which relies on subexpressions independently of each other, *Path coverage* which is based on the possible paths exercised through the code [BE90, ZHM97].
- Data-Flow coverage: In data-flow testing, a data definition of a variable is a location where a value is stored in memory (definition) and a data use is a location where the value of the variable is accessed for computations (c-use) or for predicate uses (p-use). The data-flow testing goal is to generate tests that execute program subpaths from definition to use. Traditional data-flow analysis techniques work on control flow graphs annotated with specific information on data usage [JO95, ZHM97].

Generally the functional and structural test strategies are not alternative approaches but can be used in combination because they use and provide different sources of information.

2.4 Object Oriented Testing

Object-Oriented programming (OO) has nowadays become the preferred paradigm for large-scale system design. Due to the extent of published papers and books on this argument, and specifically on the Object Oriented Testing, it is not possible to provide an exhaustive dissertation on this argument here therefore, we limit ourselves exposing the main concepts, referring the reader to [BI99] for more details.

The OO programming encompasses a body of methods, processes, and tools used to construct software systems and provides a unifying paradigm for the three traditional phases of software development: analysis, design and implementation [KM90]. It has in fact an excellent structuring mechanism, the classes, which permit the division of the system in well-defined units, which may then be implemented separately.

OO programming introduces a new concept of a subprogram, different from the traditional systems, because it attempts to separate the specification (interface) for the subprogram from its implementation (body). Hence a class can export a purely procedural interface and the internal structure of data may be hidden. This allows the structure to be changed without affecting users of the class, thus supporting software reuse and simplifying maintenance. New classes may be created as extensions of existing classes through the reuse of a class in a library, or via inheritance. In both cases the result is a reduction in the amount of software, which must be written because since previously tested classes may be utilised [KR98].

Thus, OO introduce powerful new features in the program languages, which provide visible benefits in software design and programming but also raise new problems in the Software Testing and maintenance phases. These characteristics can be summarized as [WH92, LMR92, BI99, KHG02]:

- Encapsulation: modeling and storing with an object the attributes and the operations an object is capable of performing. This increases the difficulty in controlling the object interactions and in consequently preparing suitable test cases to verify such interactions.
- Inheritance: the properties defined for a class are inherited by its subclasses, unless it is otherwise stated. However, a method that is tested to be "correct" in the context of the base class does not guarantee that it will work "correctly" in the context of the derived class. The retesting of inherited methods in a different context is therefore a rule, which increases the number of tests to perform.
- Polymorphism: is the ability to bind a reference to more then one object. This means that each possible binding of a polymorph component requires a separate test.
- Dynamic binding means code that implements an operation that is unknown until run time. These features make testing more difficult because the exact data type and implementation cannot be determined statically, and the control flow of the OO program is less transparent.

In this context the traditional Software Testing techniques, generally based on imperative programming, are often not directly applicable to the OO software with their event-driven nature. Specifically, four different levels of testing can be individuated [CCT02]:

- The algorithmic level in which the code of each operation in class is tested separately, so that conventional testing techniques can be applied;

- Class level in which the objective is to verify the integrity of a class by testing it as an individual entity.
- Cluster level is concerned with the integration of classes. As the functionality of individual classes has already been verified, the focal points are usually placed on the synchronization of different concurrent components as well as interclass method invocations.
- System level in which the interactions among clusters are tested.

Thus starting from the class level a significant difference between conventional program testing and OO Software Testing is due to the state depended behaviors. While in the former this situation is common for the embedded systems, in OO programming, many objects may have state depended behaviors and/or interact with each other. Often the subprograms are encapsulated within a larger entity, e.g. a class, working in conjunction with the other items of the same object. This situation makes the test of a subprogram in isolation very difficult because the smallest testable unit is no longer the subprogram, but classes and instances of classes [KHG02].

Another important point which differentiates the non-OO approaches from the OO is integration testing. In the former approaches such as top-down, bottom-up or big-bang can be applied. Instead in the OO context, considering a unit, the methods associated with each operation often take advantage of the underlying implementation of the class, hence testing in isolation each operation-method combination is difficult. Thus to test a class the following activities must undertake [KR98]:

a) Create an instance of the class, i.e. an object, passing the appropriate parameters to the constructor
b) Call the methods of the object passing parameters and receiving results
c) Examine the internal data of the object

This can be achieved either by writing a test program for each class and their inclusion of debug statements or by the inclusion of appropriate mechanisms in the program development environment itself. The natural testing integration is therefore to combine "subprograms" into a class, one at a time and thus testing the whole system. This avoids creating the specific object states, because they can be set by other encapsulated operations, or combinations of encapsulated operations. In this situation it is clear that specific approaches for the integration may not help since each operation may be used to test the other [BE93].

In OO programming different testing techniques can be applied starting from the cluster level as listed below. But most of them are the reviewed approaches or extension of the techniques used in the imperative programming paradigm, and only a few techniques are specific for the object-oriented context. We refer to [CCT02] for a complete description.

State-based approach relies on the construction of a finite-state machine or state-transition diagram for representing the different states of the program under test. The difficulty of the technique increases with the number of concurrent units and mainly during the dynamic instantiation of objects during program execution [KCT02].

The event-based paradigm uses temporal relationships between synchronization events such as message sequence constraints or temporal logic [CT98].

Integrated formal methods combine object-orientation with other formal paradigms such as finite state machine or process algebra [SC99, SD01].

Deterministic and reachability testing forces the synchronization to be executed in desirable orders so that a deterministic test oracle can be applied and checked with the deterministic result. Due to dynamic binding, the same synchronization may result in different binding effects at different points of input [SCK01, CLL02].

For fault-based testing, techniques such as mutation testing may not be effective in object-oriented programs and must thus readapted for this contest [KCM01]

UML-based techniques conduct testing by using the UML documentation. We provide an extensive survey of these techniques in Chapter 5.

Dynamic data flow testing helps identify anomalies of data actions by collecting information during program executions. Conventional probing techniques may not be adequate for languages that support Java-like reflection [SBA01]

2.5 Testing Measures

Measurements are nowadays applied in every scientific field for quantitatively evaluating parameters of interest, understanding the effectiveness of techniques or tools, the productivity of development activities (such as testing or configuration management), the quality of products, and more. In particular, in the software engineering context they are used for generating quantitative descriptions of key processes and products, and consequently controlling software behavior and results. But these are not the only reasons for using measurement; it can permit definition of

a baseline for understanding the nature and impact of proposed changes. Moreover, measurement allows managers and developers to monitor the effects of activities and changes on all aspects of development. In this way actions to check whether the final outcome differs significantly from plans can be taken as early as possible [FP97].

However, as stated in [PJC97] the most successful measurement program would be one in which researcher, practitioner and customer work together to meet goals and solve the problems, but this occur very rarely. The measurement process is formally defined as: *"The process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clear and defined rules"* [FP97], where an entity represents an object or an event, and an attribute is a feature or property of an entity. In other words this means representing the real world with mathematical expressions and rules so that it will be clearer and easier to understand the attributes and their relationships.

However, is not unusual to have different measures for the same thing, which can generate confusion or lead to erroneous management decisions. Thus it is important to determine which representation is the most suitable for measuring an attribute of interest i.e. the scale of measurement. As reported in [FP97] the main types of scales are:

- *Nominal*, in which the items are divided into different classes with any notion of ordering among them;
- *Ordinal*, in which the different classes are ordered with respect to the attribute; *Interval*, in which it is possible to define the concepts of distance from the ordered classes even if there is no "zero point" in the scale;
- *Ratio* in which mapping preserves the order, size of intervals, and ratios between them and where the total lack of attributes represents the zero element;
- *Absolute*, in which measurement simply counts the number of elements in the entity set.

We refer the reader to [WR01] and [IEEE98] for a wider coverage of the topic of quality measurement, including fundamentals, measures and techniques.

However, it is important to clarify that measurement is not exclusively for making predictions. The measures, or measurement systems, are used to assess an existing entity by numerically characterizing one or more of its attributes, while a prediction system consists of a mathematical model together with a set of prediction procedures for determining unknown parameters and interpreting results (i.e. predicting several attributes of a future entity). In particular validating a prediction

system in a given environment consists of establishing its accuracy by empirical means; that is, by comparing model performance with known data in given environment, while the validation of a measure ensures that this numerically characterizes the claimed attribute by showing that the represented condition is satisfied [FP97].

Considering the testing phase, it is important to clarify that there is no agreement in literature on how to classify the different applicable measures. We decide here to adopt the [BE01] classification, so we consider that measurement can be applied either to evaluate the program under test, or the selected test set or even for monitoring the testing process itself. In particular as stated in [BMB96] within each group it is possible to distinguish direct and indirect measures. To direct measures belong for example lines of code (LOC) produced, execution speed, memory size or defects reported over a set period of time. Indirect measures include functionality, quality, complexity, efficiency, reliability, and maintainability.

In the following section we report only the measures that will be mentioned in this Thesis, referring to [BE01] for a complete overview.

2.5.1 Evaluation of the Program Under Test

For evaluating the program under test the following measurement can e applied:

Program measurement to aid in planning and design testing: considering the program under test, three different categories of measurement can be applied as reported in [BE90]:

- The Linguistic measures: these are based on proprieties of the program or specification text. This category includes for instance the measurement of: Sources Lines of Code (LOC), the statements, the number of unique operands or operators, and the function points.
- The Structural measures: these are based on structural relations between objects in the program and comprise: control flow or data flow complexity. These can include measurements between program modules, in terms of the frequency with which modules call each other.
- The Hybrid measurement: these result from the combination of some structural linguistic properties.

Fault density: Generally this is a widely used measure in industrial context and foresees the counting of the discovered faults and their classification by their type. For each fault class, fault density is measured by the ratio between the number of

faults found and the size of the program [PE95]. We discuss in detail this type of measurement in Chapter 6.

Life testing, reliability evaluation: By applying the operational testing for a specific product it is possible either to evaluate its reliability and decide if testing can be stopped or to achieve an established level of reliability. In particular the Reliability Growth models can be used for predicting the product reliability [LY96]. We discuss in detail this type of measurement in Chapter 7.

2.5.2 Evaluation of the Test Performed

For evaluating the set of test cases implemented the following measures can be applied:

Coverage/thoroughness measure: Some adequacy criteria require exercising a set of elements identified in the program or in the specification by testing. In this case, during the testing the number of elements covered by test cases are monitored and the coverage (expressed in percentage) is derived as the ratio between the covered elements and the total number. The coverage can be for instance relative to the paths, the statements of the branches as well as the number of functionalities exercised during testing [PF97]. We discuss this type of measurement in Chapter 5.

Comparison ad relative effectiveness of different techniques: In this case, once established exactly what the term effectiveness means, test case are used to evaluate the effectiveness of the testing techniques applied. Possible evaluations can be the number of faults found during testing, and the improvement in reliability after testing. Analytical and empirical comparison between different techniques can be used for this [JO95]. We discuss this type of measurement in Chapters 6 and 7.

2.5.3 Managing the Test Process

For managing the test process the following measures can be applied:

Effort/Cost estimation: The testing phase is a critical step in process development, often responsible for the high costs and effort required for product release. The effort can be evaluated for example in terms of person-days, months or years necessary for the realization of each project. For cost estimation it is possible to use two kinds of models: static and dynamic multivariate models. The former use historical data to derive empirical relationships, the latter project resource requirements as a function of time [PR94]. In particular, these test measures can be

related to the number of tests executed or the number of tests failed. We discuss the cost/effort estimation in Chapters 4 and 5.

Internal vs. independent test team: An important task in test planning is the estimation of resources required which means organizing not only hardware and software tools but also people. Thus the formalization of the test process also requires putting together a test team, which can involve internal as well as external staff members. The decision will be determined by consideration of costs, schedule, maturity level of the involved organization and the criticality of the application. We discuss this topic in Chapter 4.

Summary

In this section we provide a brief description of Software Testing, presenting its objectives and various ways to achieve them. In particular, we concentrate mainly on those testing techniques and measurements which will be used and mentioned in this Thesis. Our purpose was not to exhaustively present the world of Software Testing in all its parts (other references can be used for this purpose) but to provide an orientation Chapter to the readers of this Thesis.

3 A Little Bit of Modelling Basics

Preface

In this section we provide some basic concepts concerning the Unified Modelling language (Section 3.1) and the Rational Unified Process (Section 3.2) that will be used in this Thesis, specifically in Chapters 4 and 5.

3.1 Unified Modelling Language

The definition of the Unified modelling Language, UML, started in 1994 with the cooperation of Grady Booch and James Rumbaugh at Rational Software Corporation and proceeded over the years with the collaboration of Ivar Jacobson as well until the release of Version 1.0 in the January 1997. From this date various versions have been up until the released 1.5 in March 2003.

In this section we briefly report the main concepts of UML without referring to a specific version because the intent is only to provide background knowledge useful for the reader's understanding of concepts presented in this Thesis. We refer to [UML, JBR98, RJB99] for further details.

The underlining idea of UML is to provide a modelling language for specifying, visualizing, managing and documenting the phases and characteristics of a software development process. The UML can be used for specifying requirements by creating diagrams to trace analysis and design phases (analysis and design models) and visualizing the system as assembled after its effective realization. Moreover the UML can lead the system construction in the different development phases by applying a *Round Trip approach*, i.e. using the models for code generation and reporting back the possible code modifications in the models themselves.

The UML created only as a modelling language and not as a programming language, can be used instead of textual documentation for documenting the system. The developed models in fact represent an expressive, compact and consistent way for expressing information during the life cycle.

The basic concepts of UML are summarized in three different categories: Diagrams, Views and Extension Mechanisms. We briefly discuss their main characteristic in the following subsections.

3.1.1 UML Diagram

UML diagrams are graphs describing a particular characteristic or system behaviour. UML has eight diagrams that can be used in different combinations for representing all aspects and functionalities of the system (the view of the systems). In terms of the views of a model, the UML defines the following graphical diagrams:

- use case diagram
- class diagram
- behaviour diagrams:
  - state diagram
  - activity diagram
- interaction diagrams:
  - sequence diagram
  - collaboration diagram
- implementation diagrams:
  - component diagram
  - deployment diagram

We provide a brief description of each in the following.

3.1.1.1 Use Case Diagram

A Use Case Diagram is a description of a specific aspect of system behaviour (a system functionality) and in particular represents the interactions between a number of external agents (***actors***) and their connections with the system. In particular an actor is defined as someone (a possible system user) or something (a system) that interacts with the system using information interchange

A ***Use Case*** (UC in the following) is directly connected to a requirement of the system and represents a functionality, i.e. a specific use that a system provides as perceived by an actor. In particular a UC describes the interaction between the actors and the system, not the internal logic of a system functionality, at different levels of detail (system or subsystem). The actual description of a UC is generally expressed in a textual way. In particular a UC is a class, not an instance, and describes the functionality as a whole including alternatives, errors and exceptions. An

instantiation of a UC is called ***scenario,*** and describes a specific sequence of actions that illustrates system behaviours through the interaction of the components defined in the architecture. Hence the scenarios are used to drive the discovery of use cases and actors. A Use Case is always initiated by and actor which requires a sequence of actions to the system and provides an output to an actor [EP00].

The UC and the actors are connected by ***associations*** (communication associations), which show how the actors communicate with the system. Generally an association is non-directional one-to-one relationship.

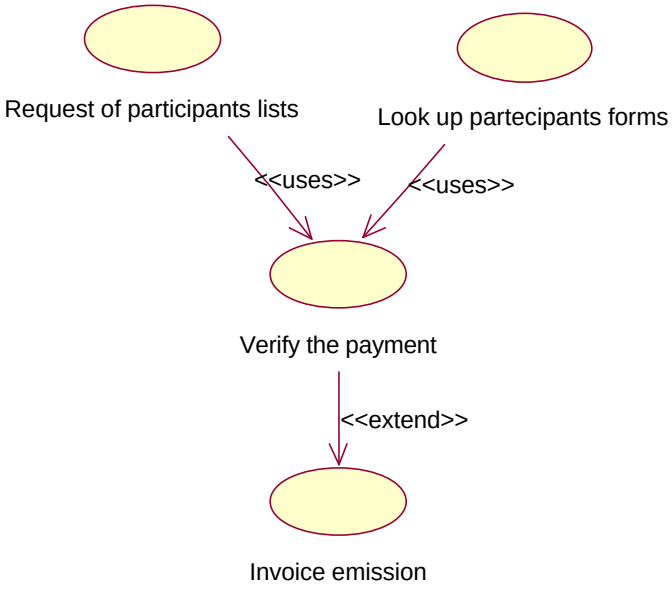


Figure 1 Relationships between UCs

A Use Case can be refined using other Use Cases, which can be put in relation to each other, by using three kinds of relationships (Figure 1):

- Extends relationship which is a generalization relationship where one use case extends another use case by adding actions to a general use case.
- Uses relationship which is a relationship where one use case uses another use case indicating that as a part of the specialized use case, the behaviour of the general use case will also be included.
- Grouping relationship when a number of use cases handle similar functionalities they can be bundled in a package

The UCs are a representation of implementation-independent system functionalities.

3.1.1.2 Sequence Diagram

A Sequence Diagram (SD) shows the dynamic collaborations between a certain number of objects highlighting the way in which a scenario is realized by the interactions of a set of objects. A SD is focused on the sequences of messages exchanged between the objects, and is characterized by two dimensions: 1) the vertical dimension represents time and 2) the horizontal dimension represents different objects [UML]. Normally time proceeds down the page and is represented by the *LifeLine* activation of the objects involved.

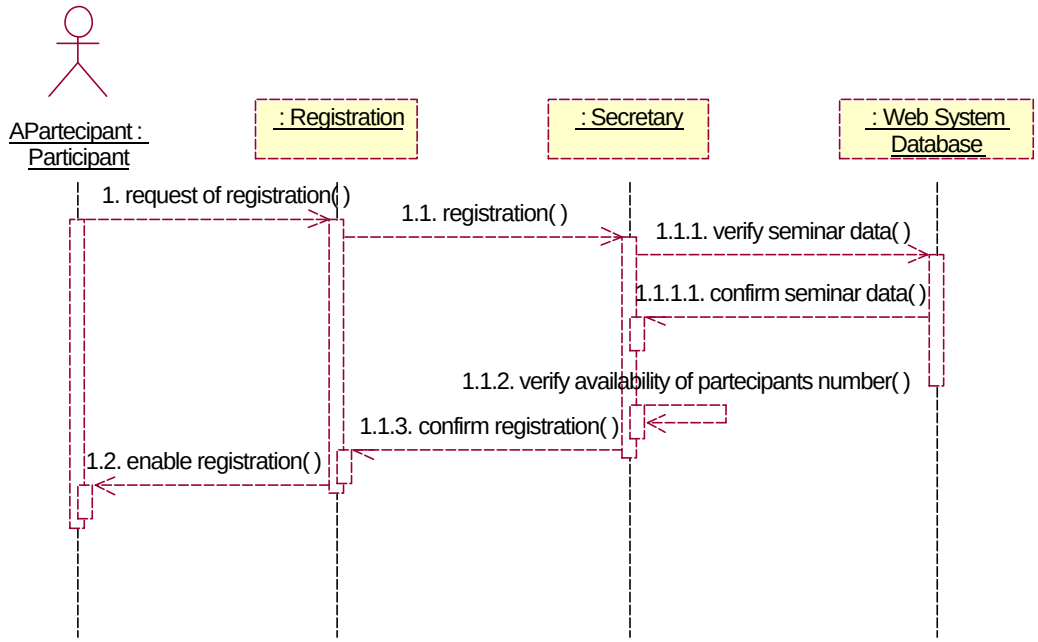


Figure 2 An example of SD

The objects exchange *Messages*, which represent the interaction between them: a sender requests a service owned by the receiver. The message activation is represented in the SD by an arrow at the head of a *focus of control region*, i.e., a rectangular box on the object Lifeline that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

There are different typologies of messages that we describe briefly below (for more details and their graphical representation we refer to [UML]):

- **Synchronous message:** when the sender who has required a service waits for the response of a receiver to continue its execution.

- **Self-delegation**: the sender sends a message to itself.
- **Asynchronous message**: the sender continues to execute after sending the message without waiting for it to be handled. It is used particularly in Real Time applications, where objects interact concurrently for creating for instance new threads, new objects, or for communicating with a thread already under execution.
- **Object deletion:** an object termination caused by another object or the object itself

The messages can then be characterized by *iterations, i.e.* when a message is sent several times to multiple receivers (ex. operation for a set of elements) or *conditions* to model branches or to decide whether or not to send a message.

3.1.1.3 Collaboration Diagram

The Collaboration Diagrams (CD) specify the objects collaborating in a specific scenario and the messages exchanged. They express the same information as the Sequence Diagrams, but while a SD describes the interactions among objects focusing on time, a Collaboration Diagram shows them in terms of space.

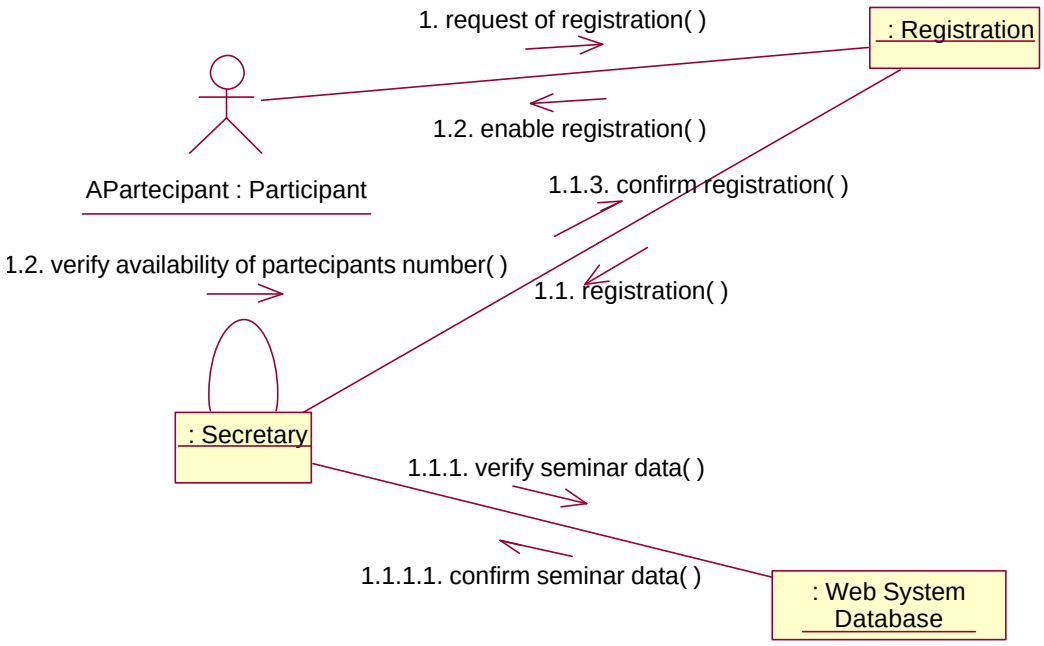


Figure 3 The CD derived by the SD of Figure 2

The CD therefore gives an explicit representation of objects' relations highlighting the collaborations among objects and emphasising more the Objects' links. In particular the messages are not ordered in space so they need an ordinal numbering. In some tools (Rational Rose, for example [RRT]) one diagram can be easily derived from the other. In Figure 3 we show the CD obtained by the SD of Figure 2.

3.1.1.4 Class Diagram

A Class Diagram shows the static structure of a system representing its classes and objects with attributes and methods. It specifies the constraints among classes using different types of associations as will be described later in this section. A system can have one or more Class Diagrams defined at diverse development phases (analysis, design and so on) representing different object typologies such as interfaces, implementation modules and subsystems.

A class in a Class Diagram is the description of an object type with its characteristics and behaviours and it is characterized by:

a) **The Class Name**: the name of the object to which the class refers
b) **The Attributes**: describe the characteristics of an object. Each attribute has a *Type* (e.g. primitive types: Integer, Boolean, Real) and a *Visibility* which indicates whether the attribute can be referenced from the other classes. The Visibility can be:
   - Public (+): the attribute can be used and viewed outside the class;
   - Private (-): the attribute cannot be accessed from other classes;
   - Protected (#): the attribute can be used only from the class or from its subclasses;
   - Implementation/Package (?): the attribute can be accessed only from classes within the same package;
c) **The methods:** describe the behaviour of a class, i.e. the actions that can be executed. They are used to manipulate attributes.

A Class Diagram consists of classes and the relationships between them. In particular the involved classes can be put in relation to each other using:

- **Association:** a class connection, i.e. a semantic link between objects of classes involved. The association is characterized by the *name*, usually a verb indicating the action, the *navigability*, i.e., the direction in which the association is

practicable (ex: a user utilizes a pc) and the m*ultiplicity* which indicates how many objects are linked to the association. Ex.: 0..1, n..m;

- **Aggregation**: a special case of association which indicates that the relation between the classes is a sort of "whole-part" (Ex. person-team).
- **Composition** (composition aggregation): a stronger concept than aggregation. The part lives inside the whole and it will be destroyed together with its whole. The multiplicity of the whole can be 0 or 1 (Ex. window-button).
- **Generalization** (inheritance): a relationship between a general and a specific class (Ex. vehicles- cars, boats). It can be used to specify a superclass i.e. a class with a more general behaviour or a *Subclass*, i.e. a class which inherits all the characteristics of the superclass (attributes and methods); it is consistent with the superclass and it specializes its characteristics.

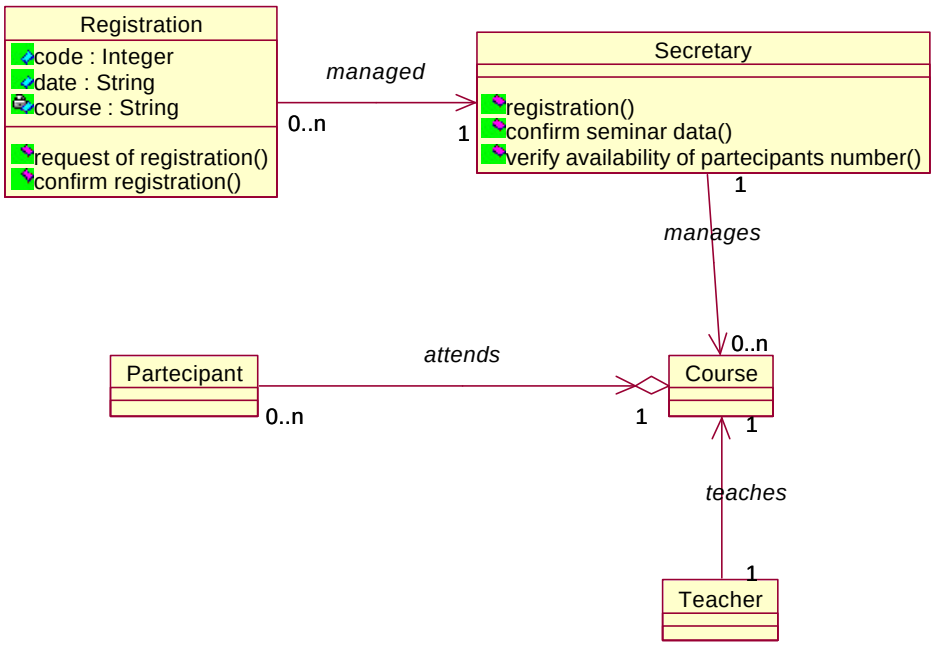


Figure 4 An example of a Class diagram

- **Refinement:** the relationship between two descriptions of the same thing but at different degrees of detail.
- **Dependency:** a semantic connection between two model elements (classes, packages, use cases) one independent and one dependent. A change in the independent element will affect the dependent element.

We show in Figure 4 an example of a Class Diagram which specifics the different relations of the object involved in the SD of Figure 2.

A specific typology of a Class diagram is the **Object Diagram** which uses the same notation and relationship of a Class Diagram, and is used to show specific links from class instances at some moment in time. It can be viewed as an example of a Class Diagram to illustrate how a complex Class Diagram can be instanced.

3.1.1.5 State Diagram

The State Diagram is the complementary description of the Class Diagram because it specifies the life cycle of the class objects. It shows all possible *states* that the objects can assume during their life and the *events* causing the state changes, called transitions. Specifically a state is the result of previous activities performed by the object and it is typically determined by the values of its attributes.

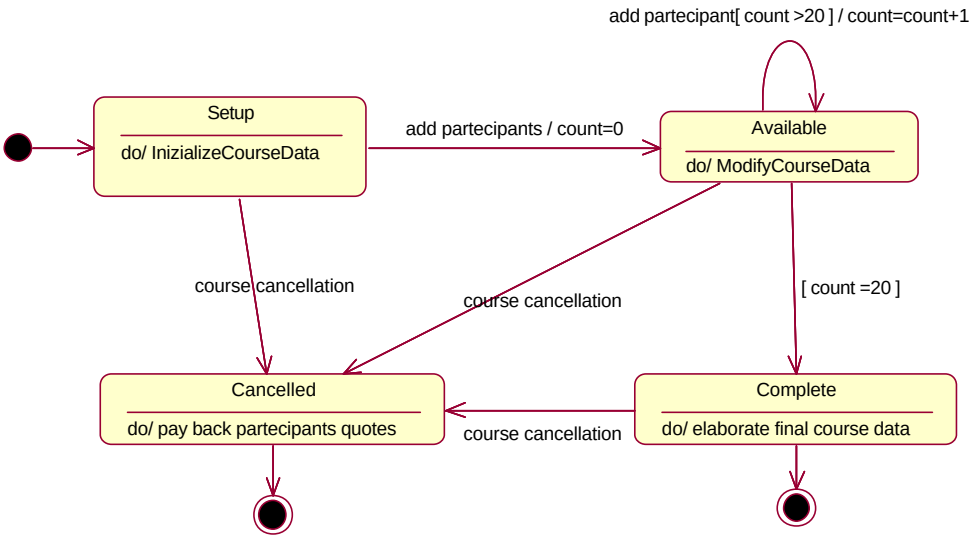


Figure 5 An example of State Diagram

There are three standard events that can determine the change of an object state: an action in the *entry* state, inside the state, *do*, and in the *exit* state. They are characterized by different types: a condition becomes true *(change event);* the receipt of an explicit signal from another object *(signal event);* the receipt of a call on an operation by another object *(call event);* the passage of a designated period of time *(time event).*

Finally a transition can be a *Parallel Transition*, i.e. it can be divided in one or more parallel transitions and the subsequent actions performed concurrently, or a *Self*

transition, i.e. the causal event brings again in the same state. We report in Figure 5 an example of the State Diagram.

3.1.1.6 Activity Diagram

The Activity Diagram shows sequences of activities, such as for instance the internal logic of a process, and it is used typically to describe the activities performed during an operation.

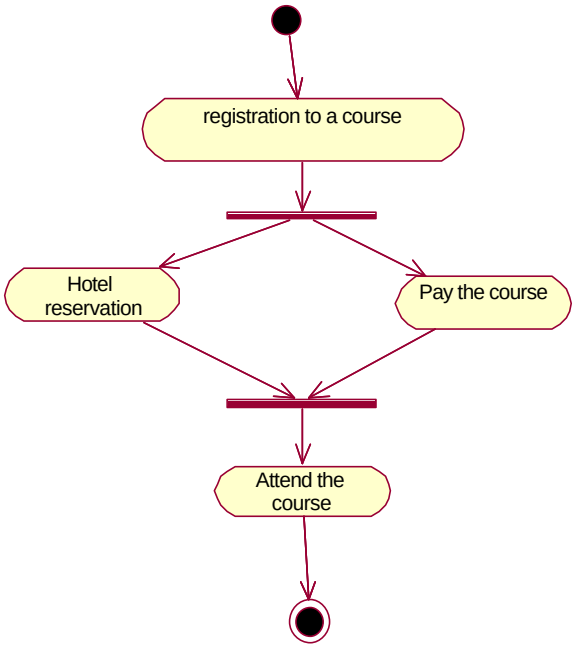


Figure 6 An example of Activity Diagram

The Activity Diagram is a variant of a State Diagram, in which the states called *activities*, allow the description of concurrency and synchronization. As in the commonly used flow-charts the Activity Diagrams describe the interactions among objects or processes showing: how actions are taken, what they do (change of object states), when they take place (action sequences), where they take place (*swimlanes*). Specifically a swimlane groups activities in vertical zones with respect to their responsibility. They are used to describe where the actions are performed (in which object) or in which part of the project. In Figure 6 we show an example of an Activity diagram.

3.1.1.7 Component/Package Diagram

A Component Diagram shows the physical structure of the code in terms of components and their dependences. In particular the components describe the implementation in the physical architecture of the concepts and the functionalities described in the logical architecture. They are executable software modules with their own identity and interfaces.

In a Component Diagram the enclosed components are characterized by dependency relations. A dependency between two components means that one component needs the other for its complete definition. In Figure 7 we show an example of a Component Diagram.

The Component Diagram can also be viewed as Package Diagram. This is a particular diagram that can be applied to any type of model element but it is usually used to collect classes and define their dependencies. Specifically every Class Diagram should be inserted in a Package diagram. In Figure 8 we report an example of a Package Diagram.

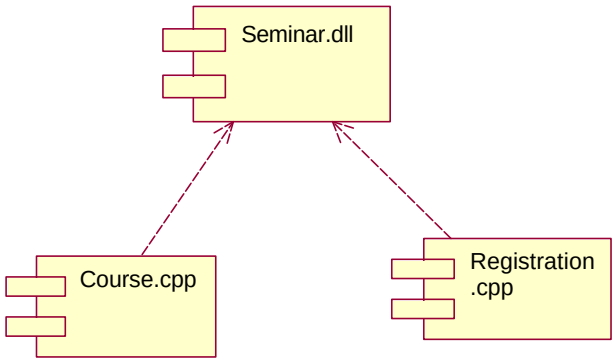


Figure 7 An example of a Component Diagram

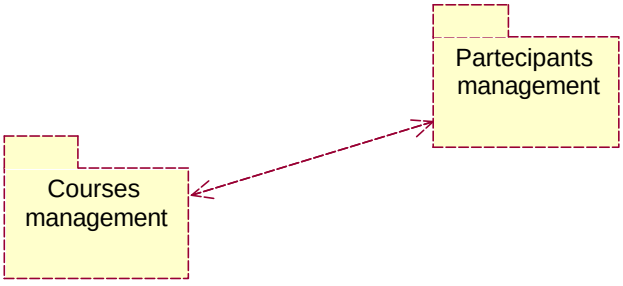


Figure 8 An example of a Package Diagram

3.1.1.8 Deployment Diagram

A Deployment Diagram depicts the run-time architecture of processors, devices, and the software components. It is the final physical description of the system's topology, describing the structure of the hardware units (nodes) and the software to execute on each unit. Specifically a *node* is a physical object (device) that has some kind of computational resource while a *connection* is the communication path among nodes. In Figure 9 we report an example of a Deployment Diagram.

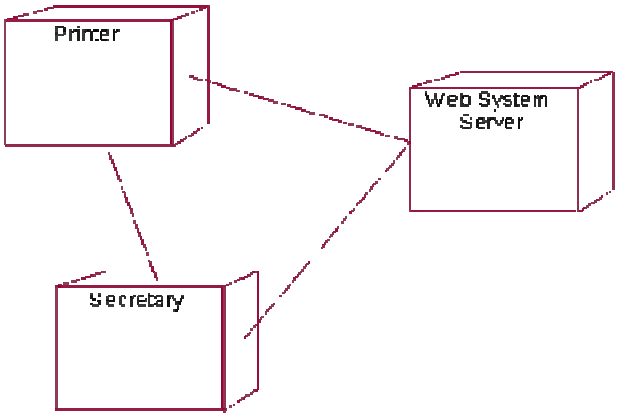


Figure 9 An example of Deployment Diagram

3.1.2 UML Views

Generally the modelling of the architecture of a (complex) system requires dealing with the problem from different points of view, considering diverse aspects such as: defining the static structure and dynamic interactions (functional aspects); establishing the timing, reliability, deployment requirements (non-functional aspects); organizing and scheduling resources and people in working groups and finally mapping to code modules (organizational aspects).

Considering the definition of the software architecture provided by the IEEE Working Group on Architecture in [IEEE01] as "the highest-level concept of a system in its environment", it is evident that architecture description requires only the definition of its structure but it also encompasses the "fit" with system integrity, with economical constraints, with aesthetic concerns, and with style. It is not limited to an inward focus, but takes into consideration the system as a whole in its user environment and its development environment - an outward focus.

Therefore the system architecture is described by a number of views *(multiple viewpoints* from OO methodologies) each representing a particular aspect of it and

addressing some specific set of concerns, specific to stakeholders in the development process: end users, designers, managers, system engineers, maintainers, and so on [RUP].

The views capture the major structural design decisions by showing how the software architecture is broken down into components, and how components are connected by connectors to produce useful forms [PW92]. These design choices must be tied to the requirements, functional, and supplementary, and other constraints. But these choices in turn put further constraints on the requirements and on future design decisions at a lower level.

Each view is described in a number of diagrams containing information that emphasizes a particular characteristic of the system. In this section we consider the typical set of views, called the "4+1 view model" [KRU95] as schematise in Figure 10. In the following we provide a brief description of each of them:

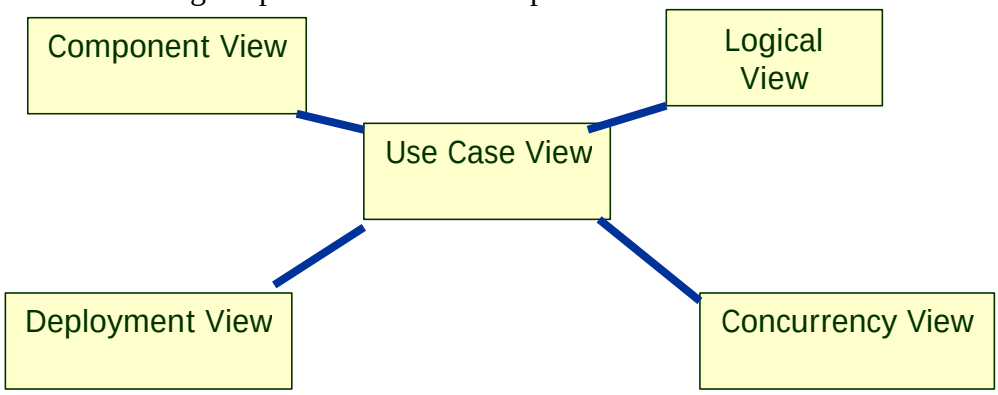


Figure 10 The 4+1 view model

3.1.2.1 Use Case View

The Use-Case View describes the functionalities the system should deliver, as perceived by external actors and contains use cases and scenarios that encompass architecturally significant behaviour, classes, or technical risks. The Use Case View is central for the development; its content drives the development of the other views, and in particular it describes the final goal of the system. Generally the Use Case view includes the following diagrams:

- Use Case Diagram
- Sequence/Collaboration DiagramActivity Diagram

Specifically, in the Use Case Diagram the UC describes a system functionality, or more precisely a requirement at different levels of detail. In the Use Case View there

could be UCs associated with main requirements, others that are related to the minor system functionalities.

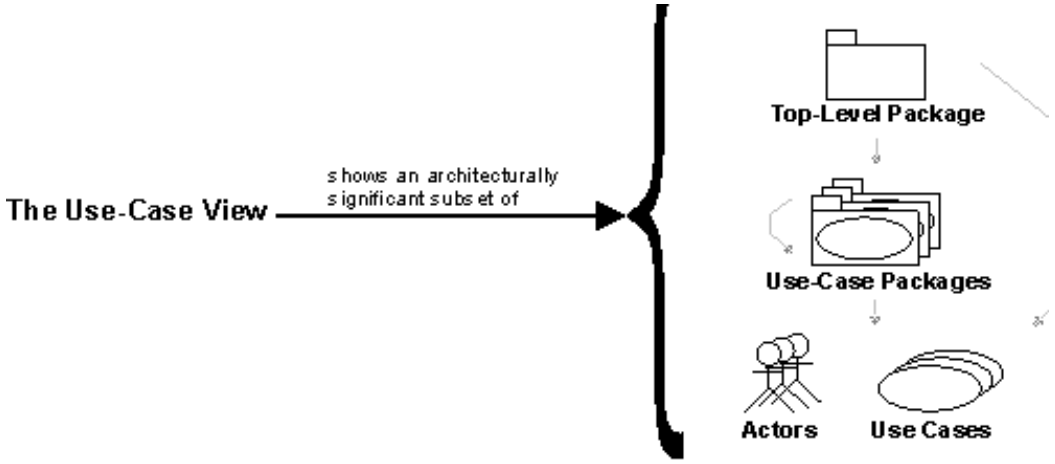


Figure 11 The use-case view as conceived in RUP.

In particular a UC associated with a high level requirement could be better specified either by using other UCs for the related subfunctionalities, or SDs and CDs for describing the required behaviour. For this reason, the UML principle for *realizing* a UC is a collaboration. It shows the implementation of the UC in terms of classes/objects and their relationships and interactions. A collaboration is represented by a number of diagrams showing the context and the integrations between the participants of the collaboration (classes/objects). The diagrams used for this purpose can be collaboration, sequence or activity

Figure 11, taken from the Rational Unified Model (RUP) documentation [RUP] shows the content of the Use Case View and where it is realized within RUP as will be better described in Section 3.2.

3.1.2.2 Logical View

The Logical View is mainly an architectural view of the system which constitutes a basis for its structure and organization. It describes how the system functionalities, depicted in the Use Case View, are realized in terms of the static structure and dynamic collaboration between objects.

The design elements of the Logical View are generally collected into packages (for instance system, subsystem, Use Case realization) and classes, possibly grouped in turn into two high level packages: the Analysis Model and the Design Model. Referring to [RUP] the former is considered an optional package, mainly representing a conceptual overview of the system. It can constitute a foundation for

the development of the Design Model, which is instead an abstraction of the implementation of the system. The Design Model represents and documents the design of the system in terms of design classes, subsystems, packages, collaborations, and the relationships between them.

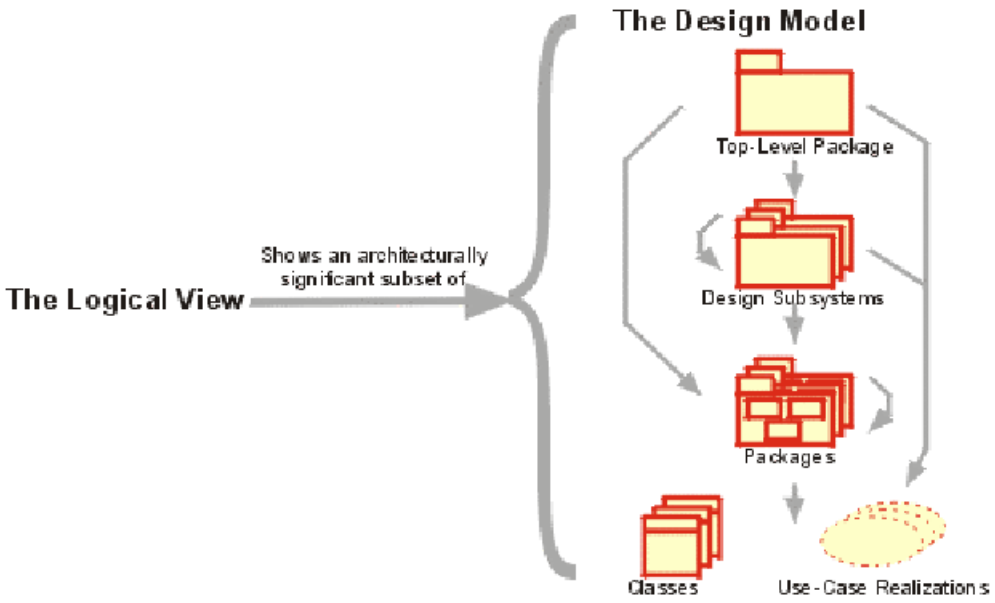


Figure 12 The Logical View as conceived in RUP

Thus the Logical View provides a basis for understanding the structure and organization of the design of the system, and generally includes the following diagrams;

- Interaction diagrams (sequence and collaboration)
- Class Diagram
- Activity and State Diagram

Figure 11, taken from [RUP] shows the content of the Logical View and where it is realized within RUP as will be further described in Section 3.2.

3.1.2.3 Component View

The Component View, called also Implementation View [RUP], is the description of the implementation modules and their dependencies. The allocation of packages and classes of the Logical View, to the packages and modules of the Component

View, is also described. The purpose of this view is to capture the architectural decisions made for the implementation. Typically, the Component View contains:

- An enumeration of all subsystems in the implementation model;
- Component diagrams illustrating how subsystems are organized in layers and hierarchies
- Illustrations of *import* dependencies between subsystems

This view is useful for assigning implementation work to individuals and teams, or subcontractors; assessing the amount of code to be developed, modified, or deleted; reasoning large-scale reuse; considering release strategies [RUP]

3.1.2.4 Concurrency View

The Concurrency View called also Process View [RUP], focuses on the division of the system into processes and processors and on the non-functional characteristics of the system used for efficient resource usage, parallel execution and the handling of asynchronous events from the environment.

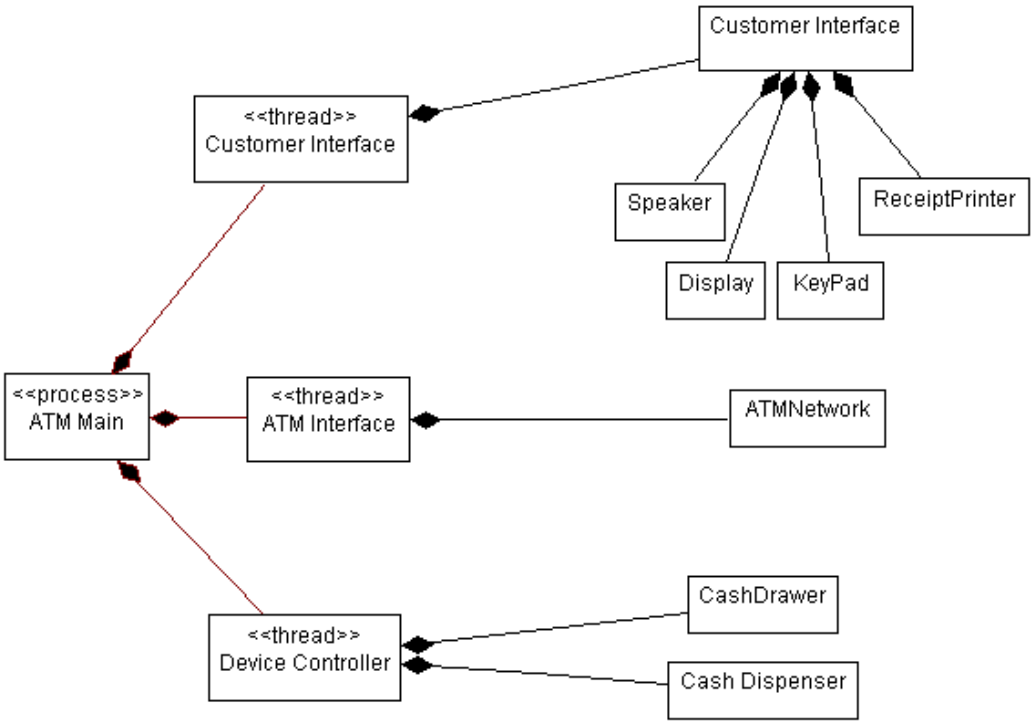


Figure 13 An example of Concurrency View for the process organization of the system.

It specifically contains the description of the tasks (process and threads) involved, their interactions and configurations, and the allocation of design objects and classes

to tasks. This view need only be used if the system has a significant degree of concurrency. As [BRJ98] states: *"With UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view - i.e. class diagrams, interaction diagrams, activity diagrams and statechart diagrams, but with a focus on the active classes that represent these threads and processes."* Of concern when constructing and using the process view are, for example, issues of concurrency, response time, deadlock, throughput, fault tolerance, and scalability. Figure 13, taken from [RUP] documentation, shows an example of Concurrency View for the process organization of a system.

3.1.2.5 Deployment View

The Deployment View shows by means of a Deployment Diagram the physical deployment of the system, such as the description of the various physical nodes for the most typical platform configurations, the allocation of tasks (from the Process View) to the physical nodes, and connection among the different nodes. This view need only be used if the system is distributed. Figure 14, taken from the RUP documentation [RUP] shows an example of Deployment View.

3.1.3 UML Extension Mechanisms

Even if the UML already provides a rich set of modeling concepts and notations, the *Meta Model,* representing the common OO concepts linked together by well-defined semantic rules, the users may require either additional features and/or specific notations or to attach non-semantic information to models.

These needs are satisfied in UML by three built-in extension mechanisms, (Constraint, Stereotype, and TaggedValue) which let the user define its own modeler's repertoire as well as attach free-form information to modeling elements. These three extension mechanisms can be used separately or together to define new modeling elements that can have distinct semantics, characteristics, and notation relative to the built-in UML modeling elements specified by the UML Meta Model.

In particular the Object Constraint Language (OCL) [WK99], which is a formal language to specify constraints and other syntax expressions related to model elements, can be used. In the next subsection we report a brief description of the extension mechanism.

3.1.3.1 Stereotype

The stereotype is the most important extension mechanisms: it defines and specializes new types of model element basing on the previously defined elements. The stereotype is therefore a semantic redefinition or extension of a previously defined semantic of the elements.

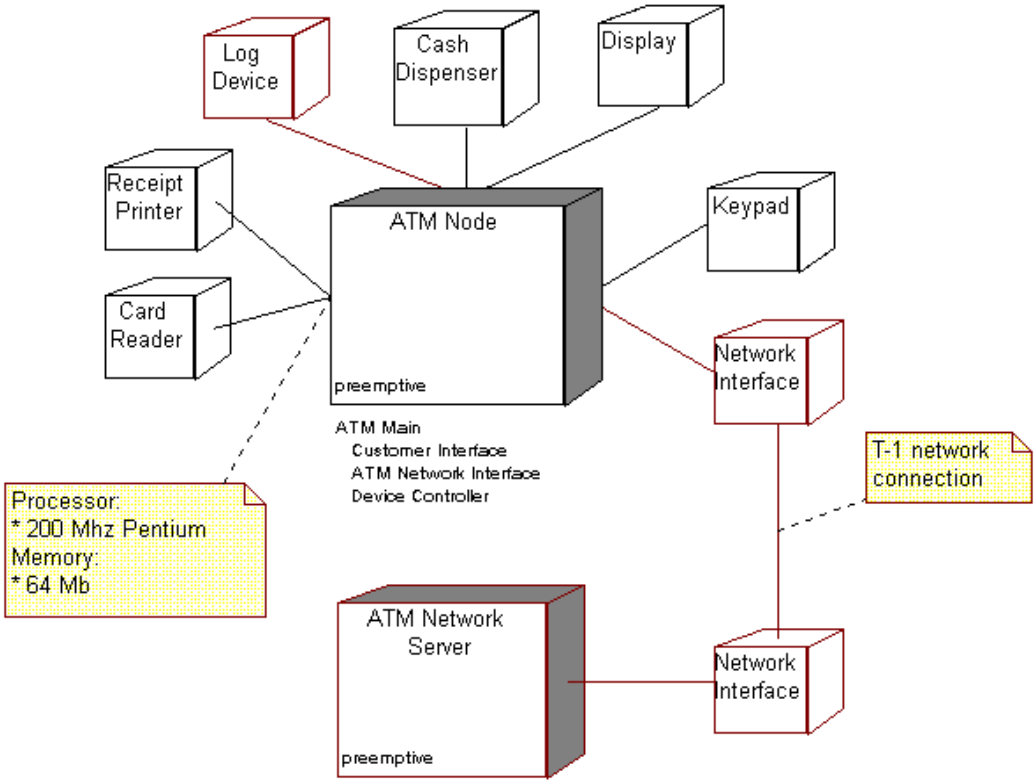


Figure 14 The Deployment View shows the physical distribution of processing within the system.

Typically it is used in classes, types, relationships, components and operations and in all the diagrams where the original element was used. Some of the standard Stereotypes are: *For classes:*

- <<actor>>;
- <<interface>>, which is described only as abstract operations linked to a class, a component or a package;
- <<control>>, <<boundary>>, <<entity>> which are used to increase the semantic meaning of the classes and their usage in modeling situations (usually Analysis Model). In particular:

- the <<boundary>> stereotype specializes the use of a class for presentation and manipulation. It presents and communicates the information in a system to another system such as human or machine and can also be used to manipulate information. Typically <<boundary>> stereotypes refer to windows, dialogs or communications classes
- <<entity>> stereotypes are used to model the core concepts.
- <<control>> stereotypes are used to connect the boundary objects with their entity objects and to handle a sequence of operations inside the system. Specifically the <<control>> stereotypes handle the processing of the information in the entity objects along with the functionality sequences that involve a number of entity objects.

For use cases (Traceability): <<use case realization>> stereotype is used to show the realization (implementation) of a system functionality described in a use case.
For generalization relationships: <<extends>> and <<uses>> stereotype
For operations: <<constructor>>stereotype
For package: <<layers>> stereotype which is used to decompose a system in groups of tasks in which each group of subtasks is at a particular level of abstraction, and <<subsystem>> stereotype.

3.1.3.2 Tagged Values and Constraints

A tagged value is a (Tag, Value) pair that permits arbitrary information to be attached to any model element. A tag is an arbitrary name; some tag names are predefined as Standard Elements as listed below. At most, one tagged value pair with a given tag name may be attached to a given model element. The interpretation of a tag is (intentionally) beyond the scope of UML, and can be shown in the diagram or separately documented. Some of the standard tagged values are: for types, invariants, for operations, Preconditions and Postconditions.

The constraint concept allows new semantics to be specified linguistically for a model element. The constraint is a semantic condition or a restriction, which can be used within the diagrams or wherever necessary. The specification is written as an expression in a designated constraint language (such as OCL).

3.2 Rational Unified Process

During the last ten years, part of software research has been dedicated to the improvement of the development process, (Software Process Improvement (SPI)

initiatives) because it realized that software products cannot be completely evaluated without also considering the process that produces them [KK00]. In this context the CMM model, [PCC93], developed at the Software Engineering Institute (SEI) is a de facto reference used by thousands of organizations together with the SPICE framework (ISO 15504) [DO99]. We report below a brief description of the commonly used process assessment models referring to [ELM01] for further details.

As summarized in [KK00], the CMM model is a framework that describes the elements required for an effective software process. In particular, it focuses on an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process. It presents sets of recommended practices in a number of key process areas that have been shown to enhance software development and maintenance capability. The CMM guides developers in gaining control of their development and maintenance processes, and evolving toward a culture of software engineering and management excellence.

Other methods for managing the program improvement are the IDEAL framework [MF96] defined at the SEI, and the Rational Unified Process (RUP), which we present in detail in this section.

As described in [KK00] The IDEAL method is an integrated approach for SPI defined by the SEI which identifies five phases: Initiating, Diagnosing, Establishing, Acting, and Leveraging. Each of these phases is centered on a particular activity:

- Initiating, which specifies the business goals and objectives that will be realized or supported
- Diagnosing, which identifies the organization's current state with respect to a related standard or reference model
- Establishing, which develops plans to implement the chosen approach
- Acting, which brings together everything available to create a "best guess" solution specific to organizational needs and put the solution in place
- Leveraging, which summarizes lessons learned regarding processes used to implement IDEAL

The Rational Unified Process [RUP], which is a detailed refinement of the Unified Process (UP) defined by Jacobson et al. [JBR98], presents itself as a Web-enabled software engineering process useful for: improving team productivity, delivering of software *best practices* to all team members, guiding the user in applying UML during the process development, and providing an extensive set of guidelines, templates, and examples. It is in particular a customizable framework,

adaptable to the different organization exigencies, supported by tools (tightly integrated with Rational tools) which automates a large part of the process development [KR00].

A central role of this process is represented by the RUP Best Practices, which are mainly guidelines for a well-established process development. RUP identify six best practices, detailed in next section, which are: Develop Software Iteratively, Manage Requirements, Use Component-Based Architectures, Visually Model Software, Verify Software Quality, Control Changes to Software.

The RUP structure is characterized by: a static structure that describes the process (who is doing what, how and when) (Section 3.2.2); dynamic structure that details how the process rolls out over time (Section 3.2.3); an Architecture-centric process that defines and details the architecture; a Use-Case Driven Process which specifies how use cases are used throughout the development cycle.

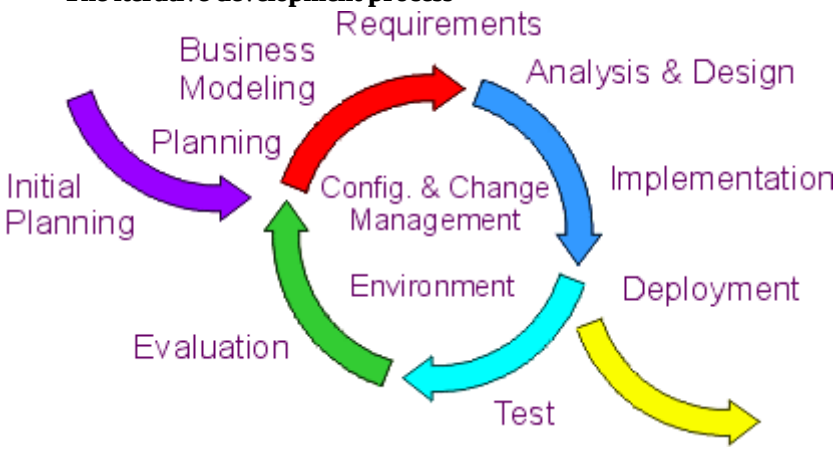
3.2.1 Best Practices

There is not a single definition of Best Practices because they treat many topics. They can be viewed as any commercially proven approach applicable to software development which, used in combination, allows identification op the root cause of software development problems [SPM]. We report a brief description of the best Practice as intended in RUP below.

- **Develop Software Iteratively**

RUP suggests avoiding the classic waterfall development process, preferring instead an iterative one. There are various reasons why it is necessary to develop the software iteratively [KR00] as a better tolerance of requirements changes, which often are the cause of project troubles, missed schedules and so on or an improvement on the integration process which are more precise.

Figure 15 The iterative development process



The elements are integrated progressively starting from the smaller ones and proceeding in a continuous and constant way, mitigating in this manner the risks as well. Moreover developing software iteratively avoids late discovery of design defects, facilitates reuse of common parts and results in a more robust product. The general schema of an iterative process is taken from [BO88] and shown in Figure 15.

- **Manage Requirements**

A requirement is defined as a condition or capability to which the system must conform. Requirements management consists of three activities: finding, organizing and documenting the system's required functionalities and constraints; evaluating changes to these requirement and assessing their impact; tracking and documenting the changing requirements of a system. Proper requirements management offers the solutions for the root causes of software development problems.

- **Use Component-Based Architectures**

A Component Base Development (CBD) permits assembly of software from manageable modules, reuse or customizing the existing components and reuse of the commercially available components. When the software is developed iteratively, by using component-based architecture, is possible to observe the continuous evolution of the system architecture. In particular each iteration produces an executable architecture that can be measured, tested, and evaluated against the system's requirements.

- **Visually Model Software**

The use of a visual notation, such as UML, allows visualizing, specifying, constructing and documenting the structure and the behavior of the system architecture. This has the beneficial effect of improving communications in the design teams and letting to hide or expose details as necessary. In particular, visual

models can be useful for many purposes: understanding complex systems; exploring and comparing design alternatives at a low cost; forming a foundation for implementation; capturing requirements precisely; communicating decisions unambiguously.

- **Verify Software Quality**

This means the continuous assessment of the quality of a system with respect to its functionalities, reliability, application performance and system performance. In particular the verification is performed by creating tests for the key scenarios, each one representing some aspect of the system's desired behavior. The management of quality has different purposes: to identify appropriate indicators (metrics) of acceptable quality; to identify appropriate measures to be used in evaluating and assessing quality; to identify and appropriately address issues affecting quality as early and effectively as possible.

- **Control Changes to Software**

The changes in software are extremely important in an environment in which multiple developers, organized into different teams, are working together on multiple iterations, releases, products, and platforms. This activity may include: definition of repeatable procedures for managing changes to software, proper resources allocation based on the project's priorities and risks; continuous monitoring of the changes including how to track, control and ensure that changes are acceptable.

3.2.2 Static Structure

The RUP static structure is presented mainly by four primary modeling elements:

- **Workers:** In RUP the term worker refers not to an individual but to the roles that must be performed to do specific work. A role is an abstract definition of a set of activities performed and artifacts owned. A worker therefore is a sort of "hat" that an individual can wear during the project. He/she performs one or more roles and is the owner of a set of artifacts. The mapping from individual to workers is performed by project manager when he/she plans and staffs the project.
- **Activity:** a specific unit of a work to be performed and is assigned to a specific worker. To each worker is associated a set of activities which expressed the workers' behavior. The granularity of an activity can be a few hours or more than one day; it may be repeated several times on the same artifact in the different iterations. In this case the repeated activities may be performed by the same

worker but not necessarily the same individual. Every activity is generally broken into steps:

- Thinking step: the worker understands the task, examines the artifacts, and formulates the outcome
- Performing step: the worker creates or update some artifacts
- Reviewing step: the worker inspects the results against some criteria

Not all the steps are necessarily performed each time an activity is invoked.

- **Artifacts:** the tangible objects of the project, which can be developed or used for producing the final product. The artifacts are used as input by workers to perform an activity and are the output, or results, of such activities. Typically they are not documents; the RUP approach discourages systematic production of paper documents, preferring to maintain the artifacts within the appropriate tool used to create and manage them. The RUP artifacts fall into five information sets: The Management set (Planning artifacts, Operational artifact), Requirement set (Vision document, Use-case model), Design set (Design model, Architecture description), Implementation set (Source code and executable), Deployment set (Installation scripts, User documentation).
- **Workflows:** are a sequence of activities that produces a result of observable value. The activities often tightly interwoven especially when they involved the same worker or individual. In UML a workflow can be expressed as a sequence diagram, a collaboration diagram or an activity diagram. There are two types of workflows: Core Workflows and Workflow Details.

  The Core Process Workflows are divided into two groups, representing a partitioning of all workers and activities into logical grouping. The first group is composed of the *Engineering workflows*: Business modeling workflow, Requirements workflow, Analysis and design workflow, Implementation workflow, Test Workflow, Deployment workflow. The second group is represented by the *Supporting workflows*: Project management workflow, Configuration and change management workflow, and Environment workflow. Each core workflow is associated with one or more models, which are in turn composed of associated artifacts, for instance use-case model, design model, implementation model, and test
- **Additional process elements:** are added to activities or artifacts to make the process easier to understand. They are divided into: guidelines, which are rules, recommendations, or heuristics that support activities and steps; templates, which

are models, or prototypes, of artifacts; tool mentors, which show how to perform the activities or steps using a specific tool; concepts, which are all the key concepts used during the process, for example, iteration, phase, risk and so on.

3.2.3 Dynamic Structure

The dynamic structure describes how the process rolls out over time, in particular in RUP, which is organized into four different phases (Inception, Elaboration, Construction, Transition Figure 16). We report in the following section a brief description of each taken from [RUP, KR00].

As shown in this figure, every phase ends with a milestone, i.e. a point in time where goals have to be reached and critical decisions must to be made. These four phases constitute a development cycle which end with software generation. Specifically the software development starts with an initial development cycle and evolves in new software generation with an evolution cycle, which can be triggered for instance by user-suggest enhancements or changes in the user's context or in the underlining technology.

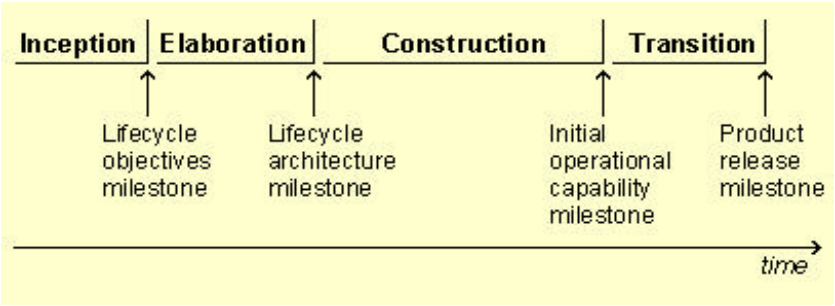


Figure 16 RUP phases

Unless the product "dies," it evolves into its next generation by repeating the same sequence of Inception, Elaboration, Construction and Transition phases. Figure 17, taken from [RUP] show the general schema of the RUP development process, demonstrating how the workflows evolve within the different phases.

3.2.3.1 Inception

The Inception phase goal is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. It is particularly important for development of new products in which there are significant business and requirements risks which must be addressed before the projects can proceed. It is important to specify that the

Inception phase is not a requirement phase; rather it is a kind of feasibility phase where just enough investigation is done to support a decision to continue or to stop.

We list here some of its main purposes:

- Establishing the project scope and boundary conditions. This includes the definition of the most important requirements and constraints, the acceptance criteria, and what is intended to be in the product and what is not.
- Discriminating the critical use cases of the system and the primary scenarios (Use Case Model).
- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios. This means mainly the evaluation of the trade-offs in design, and in make/buy/reuse, and the feasibility through simulation model or initial prototype. It is important to note that the real architecture will be realized only during elaboration and Construction phases.
- Estimating the overall cost and schedule for the entire project and the potential risks (the sources of unpredictability). This includes the evaluation of alternatives for risk management, staffing, project plan, and cost/schedule/profitability trade-offs.
- Preparing the supporting environment for the project. This means assessing the project, the organization and selection of the required tools, as well as deciding which parts of the process must be improved

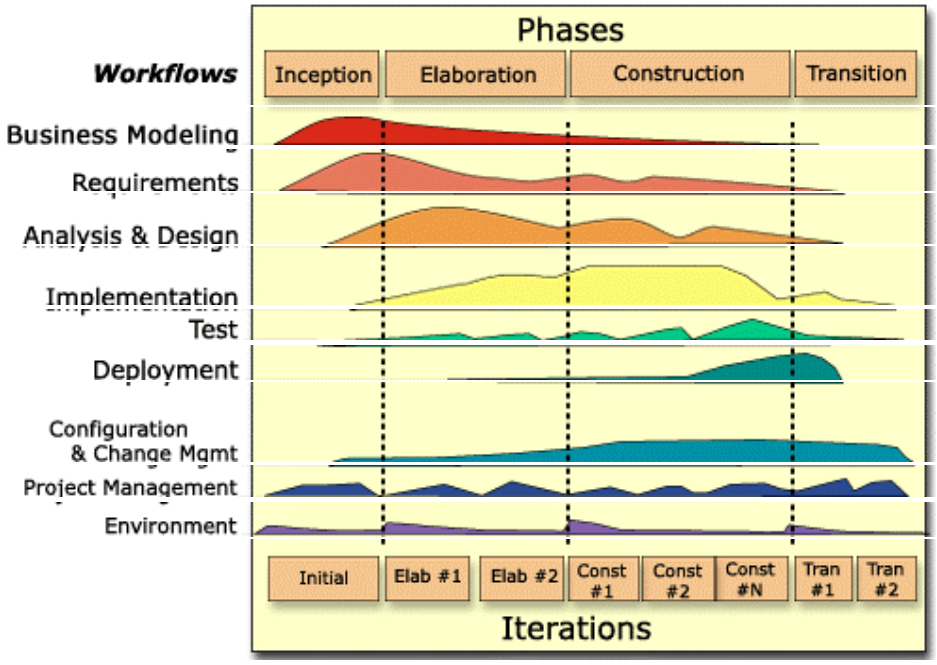


Figure 17 General Schema

For completeness we list below some of the Inception phase artifacts (in order of importance), providing a brief description for the most important. They are:

Vision document: This defines the stakeholders view of the product to be developed, and captures very high-level requirements and design constraints to give the reader an understanding of the system to be developed. Moreover, it serves as input to the Use Case Model

Business Case: This provides the necessary information from a business standpoint to determine whether or not the project is worth investing in, also establishing its economic constraints. It represents an economic plan for realizing the project, as presented in the Vision document, which must be approved at the lifecycle milestones, and updated at further milestones

Risks List: This is designed to capture the perceived risks to the success of the project and is one of the Project Manager's responsibilities to maintain and keep it updated. In particular it identifies, in decreasing order of priority, the events that could lead to a significant negative outcome and serves as a focal point for project activities and the organization of the iterations.

Software Development: This is a comprehensive, composite artifact that gathers all information required to manage the project and is again the responsibility of the

Project Manager for its completion and updating. It describes the approach for the development of the software, and is the top-level plan generated and used by the managers to direct the development effort. A key discrimi nator of a good Software Development Plan is its conciseness, lack of philosophy, and focus on meaningful standards and procedures.

Development Case: This describes the development process to follow for the individual project and it is changed based on the lessons learned at each iteration.

Iteration Plan

Use-Case Model (10-20% complete)

Prototypes

The Inception phase ends, as shown in Figure 16, with the Lifecycle Objectives Milestone which foresees the evaluation of the objectives of the project, and the decision either to proceed with the project or to cancel it.

3.2.3.2 Elaboration

The Elaboration phase is the most critical phase of each evolution cycle; its goals are to baseline the architecture of the system and provide a stable basis for the bulk of the design and implementation effort in the Construction phase. The architecture evolves considering the most significant requirements (those that have a great impact on the architecture of the system) and the assessment of risks. As for the Inception the Elaboration phase is not a requirements or design phase; rather, it is a phase where the core architecture is iteratively implemented, and high-risk issues are mitigated.

In particular, the stability of the architecture is evaluated through one or more architectural prototypes with the purpose of: ensuring that the architecture, requirements and plans are stable enough, and the risks sufficiently mitigated to determine the cost and scheduling for development completion; addressing all architecturally significant risks of the project; establishing a baseline architecture, expressed with significant scenarios, which will support the system requirements at a reasonable cost and time; producing the prototypes to mitigate specific risks such as: design/requirements trade-offs, component reuse, product feasibility.

For completeness we list below some of the Elaboration phase artifacts (in order of importance). Excluding those described in the previous phase, we provide for the most important a brief description. They are:

Prototypes: They show something concrete and executable to users, customers and managers for reducing uncertainty surrounding the stability or performance of key technology, the understanding of requirements and the usability of the product. The prototypes produced are divided into two groups, depending on what they explore and their outcome. In the former group belong the *behavioural prototypes* which focus on exploring specific behaviour of the system and the *structural prototypes*, which explore several architectural or technological concerns. To the latter belong the *exploratory prototypes* which are thrown away when done, also called throwaway prototypes, and the *evolutionary prototypes*, which gradually evolve to become the real system. The exploratory and behavioural prototypes are intended to very rapidly try out some user-interfaces and rarely evolve into resilient products.

Risk List defined previously

Development Case defined previously

Software Architecture Document: It is the responsibility of the software architect who establishes the structure for each architectural view: the decomposition of the view, the grouping of elements, and the interfaces between these major groupings. Therefore the Software Architecture Document provides a comprehensive architectural overview of the system which uses a number of different architectural views to depict various aspects of the system. In particular the use-case view must be considered before the other views, because the use cases drive the development. The process and deployment views are also considered for systems with a large degree of concurrency and distribution.

Design Model: The software architect is responsible for its correctness. He/she verifies whether the Design Model realizes the functionality described in the use-case model, and whether the architecture fulfils its purpose. It is therefore an object model describing the realization of use cases. The Design Model serves as an abstraction of the implementation model and source code and it is conceived as document for the design of the software system. It therefore encompasses all design classes, subsystems, packages, collaborations, and the relationships between them.

Implementation Model: The software architect is responsible for the integrity of the implementation model, ensuring its correctness, consistency, readability and the achievement of its purpose. The Implementation Model is a collection of the components and the implementation subsystems that contain them. Components include both deliverable components, such as executables, and components from which the deliverables are produced, such as source code file.

Vision Document defined previously

Use-Case Model: This is a model of the system's intended functions and its environment and serves as a contract between the customer and the developers. In particular it is an essential input to activities in analysis, design, and test. The Use Case Model is used by: the customer who approves it, i.e. the system is what he/she wants; the user for better understanding the system, the software architect for identifying architecturally significant functionality; the designers for getting a system overview. It is used early in the Inception phase to outline the scope of the system, as well as during the Elaboration phase. It is refined by more detailed flows of events during the Construction phase.

The Elaboration phase ends, as shown in Figure 16, with Lifecycle Architecture Milestone which foresees the examination of the objectives and scope, the choice of architecture, and the resolution of the major risks. The project may be aborted or considerably reconsidered if it fails to reach this milestone.

3.2.3.3 Construction

The goals of the Construction phase are to clarify the remaining requirements and completing the development of the system based upon the baseline architecture. This is mainly a manufacturing process, where emphasis is placed on managing resources and controlling operations to optimise costs, schedules, and quality. In particular the Construction phase is a transition from the development of intellectual property during Inception and Elaboration, to the development of deployable products during Construction and Transition. The main activities of this phase are:

- Minimizing development costs by optimising resources and avoiding unnecessary scrap and rework,
- Achieving adequate quality and useful versions (alpha, beta, and other test releases) as rapidly as is practical,
- Develop iteratively and incrementally a complete product that is ready to make the transition to its user community. This includes the description of the remaining use cases and other requirements, the completion of the implementation, and the testing of the software to decide if the software, the sites, and the users are all ready for the application to be deployed.

We list below some of the Construction phase artifacts (in order of importance). Excluding those described in the previous phases we provide a brief description for the most important. They are:

Deployment Plan: This describes the set of tasks necessary for installing and testing the developed product so that it can be effectively transitioned to the user community and provides a detailed schedule of events, persons responsible, and event dependencies. The Deployment plan is begun in the Elaboration phase and is refined in the Construction phase.
Implementation Model: defined previously.
Test Suite: The Test Designer is responsible for this artifact, who has two sets of responsibilities. The primary set consists in implementing each Test Suite according to defined standards; identifying opportunities for reuse and simplification; managing all subsequent changes to it. The secondary set consists in identifying for each Test Suite its needs and the requirements, and ensuring that the Test Suite encompasses a collection of test cases that are useful to validate together. A package-like artifact is used to group collections of Test Scripts. Sometimes these groups of tests can refer directly to a subsystem or other system design element; other times they relate directly to quality dimensions, requirements compliance and so on.
Training Materials: They refer to the material that is used in training programs or courses to assist the end-users with product use, operation and/or maintenance. The purpose is to teach users how to use, operate or maintain the product. The training Materials are needed if there will be formal education of users or system operations staff.
Design Model: defined previously.
Development Case: defined previously.
Data Model: This is a subset of the implementation model which describes the logical and physical representation of persistent data in the system. It also includes any behaviour defined in the database, such as stored procedures, triggers, constraints, and so on. The Data Model is specifically needed where the persistent data structure cannot be automatically and mechanically derived from the structure of persistent classes in the design model

The Construction Phase ends, as shown in Figure 16, with the Initial Operational Capability Milestone. At this point it is necessary to decide whether the software, the sites, and the users are ready to become operational without exposing the project to high risks. All functionality has been developed and all alpha testing (if any) has been completed (see Chapter 2). In addition to the software, a user manual has been developed where there is a description of the current release.

3.2.3.4 Transition

The focus of the Transition phase is to ensure that software is available for its end users. The Transition Phase can span several iterations, and includes testing the product in preparation for release, and making minor adjustments based on user feedback. By the end of the Transition Phase the project should be in a position to be closed out. In some cases, the end of the current life cycle may coincide with the start of another lifecycle on the same product, leading to the next generation or version of the product.

For completeness we list below some of the Transition phase artifacts (in order of importance). Excluding those described in the previous phases for the most important we provide a brief description. They are:

Product Build: This is the packaging of a product for market. In particular a product can contain multiple deployment units, and may be accessible as a downloadable commodity, in shrink-wrap or on any digital storage media formats. The Product is defined as a Deployment Unit that has been packaged for sale and distribution. Typically the product is released to manufacturing in the late Transition iterations. By that time the software has undergone internal and beta testing, and is sufficiently mature for mass production.

Installation Artifacts: They refer to the software and documented instructions required to install the product. These artifacts are needed if installation programs will be used to configure the system in the deployment environment. If the software is deployed only once (as is the case with many systems built by a company for internal use on a corporate server), installation artifacts may be omitted. In a system where the end user is expected to install the product, the Installation Instructions can be included in the user's guide.

Training Material defined previously

End-User Support Material: It consists of the user manual and provides instructions for using the software. In particular it provides the basis for test plans and test cases, and for construction of automated test suites. This is typically required for any system that has a user interface; systems that have little or no user interface may omit it.

The Transition phase ends, as shown in Figure 16, with the Product Release Milestone. At this point, it is necessary to decide if the objectives were met, and if they should start another development cycle. In some cases this milestone may coincide with the end of the Inception phase for the next cycle. At the Product

Release Milestone, the product is in production and the post-release maintenance cycle begins. This may involve starting a new cycle, or some additional maintenance release.

Summary

We have provided in this section basic knowledge about both UML and RUP, necessary for the comprehension of the methodologies presented in this Thesis.

PART 2:
A SOLUTION FOR TEST PLANNING MANAGEMENT

4 The Propean Approach

Preface

In this Thesis we consider the overall testing process starting from its initial stages, i.e. the definition of the Test Development Plan including the resource estimates (specifically time, staff, and development environment costs in particular) up to the effective Test cases execution.

In this Chapter we start our journey in the testing phase, discussing an original methodology, the Propean approach, useful for Test Development Plan definition. Specifically the Propean intent is to propose a valid and reliable solution to the managers to support the decision-making process in a multiproject management environment.

From this perspective, in this Chapter we describe the Propean approach, Section 4.4 and the application of Propean to support the decision to release a product, based on the analysis of trouble reports (Section 4.5).

However, the use of Propean methodology is not limited only to testing management, but can be adopted in every development phase as well as for the organization of the entire development process for defining the Software Development Plan. In Section 4.6 we show the application of Propean to a case study encompassing the modelling of the entire Rational Unified Process.

4.1 Propean Scope

Planning the testing phase is a difficult and critical task for project managers, which requires evaluating whether the resources assigned to a specified task are adequate or whether under the existing organizational schemes the predicted time schedules will be met. Making such decisions is very complicated, because the processes involved are highly complex: the influencing factors (both human and technical in kind) are many, and in most cases not easily measurable or predictable.

We present the Propean methodology, Project Performance Analysis [BMM03], which relies, as the name implies, on Software Performance Engineering (SPE)

[SM90, SW01] and queueing networks models [LA83]. Specifically following the metaphor that:

- *Project teams* correspond to the *processing resources* in performance models,
- *Project activities* are the *tasks* to be performed within established time intervals.

We readapt in Propean the performance analysis methods for: assessing the time to completion of specified testing activities or the overall testing phase, handling personnel multitasking during different projects, optimising the workloads distribution.

For modelling the testing phase and obtaining the required estimations we embrace the trend of using UML as input modelling notation [FR99, JSW99, EP00, MA00, NLS02] (specifically we adopt the standard RT-UML specialized profile [UMLP]) and performance techniques for system evaluation [WO00, WO02, CM02]. As a result we obtain an integrated approach that allows the managers to: use familiar notations and tools to define models of the flow of testing activities to be performed and of the tasks to be distributed among personnel; express their expertise by tuning the input models with the proper parameter values; automatically derive measures of interest which rely on a solid mathematical background and have a statistical validity.

Actually the idea of using performance techniques in project management is not completely new, but applications so far have been limited to a single project at a time, as for example in [AMN95], or have been developed to handle specific situations, as in [ACL01] for simulating the performance of geographically distributed cooperating maintenance service centres, and not as a general approach. In contrast, Propean provides a generic solution which can handle multiple projects and can be applied to any situation and workflow of activities.

4.2 Related Works

Extensive literature about project management and development can be found, but little of it treats the problem of multiproject planning and people multitasking on several parallel projects. In this context the Software process simulation modelling is one of the widespread techniques used for strategic management of software development, supporting process improvement and training of software management [KMR99]. In particular a software simulation model represents some specific aspects either of the current implemented or of the hypothetical future implementation of process. We report here a brief survey of major related studies and of the more

widespread decisional tools. We refer to [KU01] for a more complete review of the research on the decision in product development and to[KMR99] for an introduction of the software process simulation modelling.

Two crucial aspects of project management during development are resources distribution and activity planning. These issues belong to a more general research field which is Concurrent Engineering (CE) [SM97]. This discipline became popular with the studies of Imai et al. [IKT85] and Takeuchi and Nonaka [TN86] and has greatly influenced both the academic and the industrial approaches to production. However, these works focus mainly on organizing tasks within a single project, taking into account the decomposition of a complex product design into smaller activities and their subsequent coordination.

Considering the distribution of resources in a multiproject environment, PERT (Project Evaluation and Review Technique) [KA86] and CPM (Critical Path Methods) [DE85] are probably the first proposed methods. They describe an idealized flow of project activities, in which no new project is introduced over time and activity durations are treated as deterministic. Markov chain models [KU01, WE86], which assume an exponentially distributed activity time and use matrix methods for deciding the task time order in development [BFS90] were the natural subsequent evolutions.

The work presented here is close to that of Adler et al. [AMN95]. These authors study the problem of personnel organization and resources distribution among several projects developed at the same time, and like us use queueing networks and stochastic processing network models to represent product development and identify the bottlenecks in task scheduling. However the authors focus on five basic process elements: jobs, tasks, procedure constraints, resources, and flow management control. In particular, a single process may need to handle a variety of job types, which in turn are divided into tasks (i.e., activities or operations). Tasks are connected by precedence relations. The resources are engineers and technicians, who are the units that execute the tasks. The flow management control represents how the resources executed a job's constituent tasks. Lock [LO98] identifies a sixth element consisting of the assessment of individual contributions.

Recently queuing theory has been applied to model requirement management [HRN01], software maintenance requests [PMB99], [WC99], [RA00] and to management planning [ACL01]. Specifically, the latter case presents a queueing-based approach for staffing process management and evaluating service levels. The

nodes of a multi-stage, multi-centre queueing model are associated with the different maintenance phases. Each stage is considered in series and each entering request is subjected to a sequence of activities before leaving the system.

4.2.1.1 Decisional Tools

The decisional support that managers can use is generally of two kinds. One consists of techniques or methods that visualize resources and personnel and distribute them among the phases of project development. Examples are represented by the traditional Control Charts or Gantt Charts [BO96], or the more innovative Design Structure Matrix (DSM) [BR01] which can display the interactions between different teams with the process activities. Tools may support these methods, which are extremely intuitive, but generally the validity of the plans depends strictly on the subjective skill of the managers. Besides, the use of these techniques in a multiproject context could be rather difficult.

The second kind of decisional support consists of specialized tools for managers. Microsoft Project Tool [MPT] or the Kerzner Project Management Maturity Online Assessment Tool [KPM] represent some examples of specific tools which provide a valid aid in maintaining an updated database of available people and resources, and for producing and visualizing a project plan.

Recently, the idea of readapting existing tools for management purposes is becoming more common for economic aspects as well and some proposals can be found in the literature. An example is the work of Dickinson et al. [DTG01], which shows how to use Dependency Matrix in combination with the existing Portfolio tools to support the decisional process, analyse the interdependences between projects and combine them. Another solution is presented in [BLP01] where the authors propose a tool for production management optimisation using Gantt Charts and PERT diagrams for visualizing the obtained results.

However, most existing tools consider only a specific aspect of management, focusing for example either on the completion time or on personnel distribution and, more importantly, they cannot explicitly manage several contemporaneous projects. Finally, the majority of available tools apply ad hoc algorithms for simulating project evolution, based on some parameter values introduced by the user. Some of those tools generate approximate predictions without any guarantee of statistical significance.

Thus the approach presented here attempts to overcome the mentioned limitations of the existing tools, proposing an innovative solution for project management.

4.3 Background Knowledge

In this section we briefly report the background information necessary for understanding the Propean methodology. In particular in Section 4.3.1 we present the basic concepts of performance engineering, in Section 4.3.2 the RT-UML profile and specifically the Performance Analysis Profile.

4.3.1 Basic Concepts of Performance Engineering

We provide here the definitions and the basic concepts of performance engineering used in the development of the Propean Methodology, without aiming to supply a complete documentation.

Generally the application of performance techniques has, as its main objective, the quantitative evaluation of the system under development. Specifically its performance can be expressed in different ways including: response time, throughput, or constraint on resource usage [SM90]. The response time is typically one of the main elements characterizing the quality of a system and is described from a user perspective, for instance the number of seconds it takes to respond to a user request. It is conditioned by a number of factors such as the execution time of the device, the entity of the requests, and the number of simultaneous users. This last factor in particular mainly complicates the evaluation of the response time due to the management of the queue of requests for the same device.

Therefore, a performance model must be developed for evaluating the performance of a system. Solving this model much information can be derived such as the mean response time and the identification of devices representing bottlenecks for the system performance.

Different approaches can be used for generating a performance model but we only consider those based on the Software Performance Engineering (SPE) presented first in [SM90]. This is a systematic and quantitative approach for constructing software systems, which is based on the careful and methodical assessment of performance issues throughout the lifecycle, from requirements and specification to implementation and maintenance. The SPE process includes the following steps [SW01, SE02]:

1. *Assess Performance Risk*: establish the effort to put into SPE activity, i.e., the level of risk and its impact on system performance
2. *Identify Critical Use Cases*: determine which use cases are most important either for operation of the system or for responsiveness or scalability for the user(s) of the system
3. *Select Key Performance Scenarios:* identify for each use case the most important scenarios, i.e., those which are executed frequently or that are perceived as critical to the performance
4. *Establish Performance Objectives*: for each key performance scenario specify quantitative criteria for evaluating its performance characteristics and the conditions (workload mix and intensity) under which the performance objective should be achieved
5. *Construct Performance Models:* explained in detail below
6. *Determine Software Resource Requirements*: identify the amount of processing and software resources required for each scenario step
7. *Add Computer Resource Requirements:* include the resources and devices to be used by scenario steps. Computer resource requirements depend on the environment in which the software executes.
8. *Evaluate Performance Models:* using the model and the selected analysis method, compute the performance predictions. Whether there are not problems, proceed to solve the execution model. Otherwise two alternatives are possible: modify the product concept choosing the most promising design approach and evaluate the effect on performance; or revise the performance objectives to adapt them to the new reality
9. *Verify and validate the models:* these activities proceed in parallel with the construction and evaluation of the models themselves and have the purpose of verifying the accuracy of the predictions

Considering the construction of the performance model (activity 5), the SPE basic concept is the separation of the *Software Model* (SM) from its environment (i.e., hardware platform model or *Machinery Model*, MM). This distinction, on the one hand, allows for defining software and machinery models separately and solving their combination, on the other improves the portability of the models (e.g., the performance of a specific software system can be evaluated on different platforms, and the performance of a specific platform can be validated under different software systems).

The SM captures the essential aspects of software behaviour; we represent it by means of *Execution Graphs* (EGs) (Appendix A). An EG is a graph whose nodes represent software workload components and whose edges represent transfer of control. A software workload component can be a single instruction or a whole procedure, depending on the granularity adopted for the model [SM90]; this feature makes EGs suitable for modelling software at different levels of detail.

EGs include several types of nodes (or blocks), such as basic, cycle, conditional, fork and join nodes. In Appendix A we give a brief description of each of them while Figure 6 and Figure 7 show examples of EG. Each node is weighted by use of a demand vector representing the resource usage of the node (i.e., the demand for each resource).

The MM model is the hardware platform and is based on the *Extended Queueing Network Model* (EQNM) [LA83]. To specify an EQNM, we need to define: the components (i.e., service centres), the topology (i.e., the connections among centres) and some relevant parameters (such as job classes, job routing among centres, scheduling discipline at service centres, service demand at service centres). Component and topology specification is performed according to the system description, while parameters specification is obtained from information derived by EGs and from knowledge of resource capabilities. In particular an EQNM is characterized by nodes and arcs, which connect the nodes. To in case of branching to each arc is associated value, called routing probability, representing the probability that a job will cover that path. Obviously the sum of the values associated to the outgoing arcs of a branch must be equal to 1. In Figure 8 an example of an EQNM is reported, while in Appendix A we discuss the Queuing network in detail.

Once the EQNM is completely specified, it can be analysed by using of classical solution techniques (simulation, analytical technique, hybrid simulation [LA83]) to obtain performance indices such as the mean network response time or the utilization index.

4.3.2 RT- UML: the Performance Analysis Profile

Although UML is generally recognized as a useful tool for modelling the functional characteristics of a system (e.g., see papers in [UML00, UML01, UML02]), historically it had ignored non-functional requirements, such as response time, availability, throughput and bandwidth. These constitute important system

features, nowadays often referred to in abstract as the QoS (Quality of Service) characteristics.

By general consensus the UML lack of a quantifiable notion of time and resources was felt to be *"an impediment to its broader use in the real-time and embedded domain"* [SE01]. As reported in [SE01], in 1999 to cope with the needs from this key area, the Analysis and Design Platform Task Force of the OMG issued an explicit request for proposals (RFP) for a UML domain-specific interpretation (to be fully conformant with the UML standard) capable of dealing with non-functional requirements.

In response to the OMG RFP, a working consortium of OMG member companies proposed a UML Profile for Schedulability, Performance and Time (RT-UML), which has been recently adopted as an OMG standard profile [UMLP].

Presenting a detailed overview of the RT-UML profile is beyond the scope of this Thesis; we provide here only the essential background necessary for understanding the RT-UML features we use in the Propean methodology. For major details we refer the reader to [UMLP].

RT-UML is not an extension of the UML metamodel, but a set of domain profiles for UML allowing for the construction of models that can be used to make (early in the life cycle) quantitative predictions regarding the characteristics of timeliness, schedulability, and performance. In particular, effort has been spent both to enable predictive quantitative analyses (e.g., the ability to determine the schedulability of a planned piece of software or its response time), and to model QoS aspects, such as deadlines and priorities.

The idea underlying the RT-UML is to import, as annotations in the UML models, the characteristics relative to the target domain viewpoint (performance, real-time, schedulability, concurrency), in such a way that the various (existing and future) analysis techniques can usefully exploit the provided features.

Generally domain viewpoints are not often used in practice because they require great expertise and specialized knowledge. The intent of RT-UML profile is to overcome this problem by providing a single unifying framework to encompass the existing analysis methods, while leaving enough flexibility for different specializations. At the core of the profile is the *general resource modelling* framework, which provides a common model of resources and of their QoS attributes. Then, based on this common framework, more specific sub-profiles are defined, i.e., "profile packages dedicated to specific aspects and analysis techniques".

Their purpose is to specialize the generic concepts to better represent the needs of a specific domain, i.e., to derive a *conceptual domain model.*

The general resource modelling framework itself consists of three sub-profiles dealing respectively with resource modelling, concurrency and time-specific concepts. In the next section, we focus in particular on the RT-UML sub-profile we use in Propean, i.e., the Performance Analysis (PA) profile.

The PA profile is specifically designed for capturing performance requirements and specifying the QoS characteristics or execution parameters. At a high level of abstraction, the concepts characterizing the PA profile are:

- The *scenario*s, i.e., ordered sequences of steps, describing various dynamic situations involving the use of a specified set of both processing and passive *resources* under specified *workloads* (i.e., the load intensity and the required or estimated response times for the scenario). In particular we can distinguish between: a *closed workload,* in which a fixed number of requests cycles while the scenario is executed, and an *open workload,* in which the requests arrive at a given (predetermined) rate.

A step in a scenario is characterized by its mean execution number (i.e., the mean number of times it is repeated when executed) and the host execution demand (i.e., the execution time taken on its host devices) and might involve multiple concurrent threads, due to forking.

- The *resources*, i.e., the servers in a performance model that can be active or passive. The active resources are usually servers characterized by the *service time*, i.e., the execution demand of the steps that are hosted by resources, while the passive resources can be acquired and released during scenarios and are characterized by the holding times.
- The *performance measures* of the system that include: resource utilizations, waiting times, execution demands, and response times. Each of these values can be: derived from the system requirements or performance constraints (e.g., response time for a scenario); estimated on the basis of experience or previous knowledge (e.g., execution demand); directly measured or simulated.

The RT-UML PA profile provides UML extensions to deal with the above notions of scenarios, resources, and workloads and the associated attributes (in the following, PA attributes), so as to allow for extensive and wide-ranging performance analyses. In our methodology, we are actually interested only in a small subset of these extensions.

PA scenarios can be modelled following either a Collaboration-based approach or an Activity-based approach (as in [PS02]). In the tradition of [CM02], we take here the former approach, and represent a scenario by an annotated Sequence Diagram. The use of Activity graphs might present some advantages in expressiveness [UMLP] (modification of the Propean approach to allow usage of Activity graphs is part of our future plans).

We report below a short description of the subset of PA annotations we use in Propean. They concern the workload, the steps and the resources involved in the scenario considered. For each annotation we specify the associated stereotype, the attributes and the UML extensions (PA attributes) used for representing these domain concepts (for more detail see [UMLP). In particular:

- **closed workloa**d: a fixed number of jobs cycles indefinitely in the scenario, and spends an external delay period. The stereotype used is <<PAclosedload>>
  - Attributes and associated PA attributes:
    - *Population:* the size of the workload, i.e., the number of jobs involved (PApopulation)
    - - *Response time:* the delay between the instant in which the scenario starts and that in which it is completed (PAresptime)
- **Step:** each increment in the execution of a scenario that can involve the use of resources is a step. The granularity of a step depends on the level of abstraction associated with the scenario. The stereotype used is <<PAstep>>
  - Attributes and associated PA attributes
    - *Repetition:* the number of times the step is repeated (PArep)
    - *HostExecutionDemand:* the total execution demand of the step on its host resources, i.e., the service demand necessary for accomplishing the request (PAdemand)
- **Resource:** This can be passive or active and can participate in one or more scenarios. The former is generally protected by an access mechanism and can represent either a physical device or a logically-protected access. The latter can be a processor, an interface or a storage device and is characterized by the processing steps allocated to it along system deployment. The stereotype used is <<PAresource>>
  - Attributes and associated PA attributes
    - *Utilization:* this is usually the result of an analysis and represents the computed utilization of processing resources expressed as a percentage.

For a passive resource in particular it represents the mean number of concurrent users of the resource (PAutilization)

- *SchedulingPolicy:* the policy that controls the resources, i.e., the rules for assigning the resources to a set of steps (active resource) or the access control policy for handling requests from scenario steps (passive resource). The scheduling policy can be for example FIFO (first-in- first-out), PS (processor sharing), LIFO (last-in-first-out) and so on (PASchdPolicy)
- *ProcessingRate:* (only for active resources) the relative speed factor of the processor, expressed as a percentage of some normative processor (PArate)
- *IsPreemtable:* (only for active resources) the possibility for the resource to be preemptable or not, once it starts the execution of an action (PApreemptable)
- *Throughput*: the rate at which the resource performs its function (PAthroughput)

The numerical values associated with the PA attributes may have different interpretations; for example, they may represent a fixed value, a variable to be estimated, an average value or a distribution, or else they may be a prediction, a measure or a requirement. To model PA value semantics, RT-UML follows a predefined syntax, whereby it is possible to specify all the desired characteristics (for an example of application see Section 4.5).

4.4 The Method

The method presented in this section provides for the manager a sound, reliable solution supporting the decisional process in multiproject management, by the use and readapting of the techniques of Software Performance and queueing networks[1]. In particular starting from the metaphor of Section 4.1 and using the above SPE concepts, we capture in the SM those aspects relative to the activity planning, while in the MM those relative to people (over/under) utilization and distribution.

[1] These are in fact the most widespread methods in the performance field, but the application of other approaches, like Petri nets [LI98], LQNs or process algebras [HR01] could be used instead, by applying the appropriate transformation rules from the UML diagrams to these notations [UML00, UML01, UML02, WO00, WO02].

In particular, we apply the method proposed in [CM02] extended to the RT-UML profile (Section 4.4.1), for the derivation of performance models based on SPE techniques, starting from a set of UML diagrams. Precisely, the SM is derived from a Sequence Diagram (SD), and the MM from a Deployment Diagram (DD). The method then extracts from these diagrams the main factors affecting system performance and combines them to generate a performance model.

In Figure 1 we outline the basic steps of the Propean methodology (and who is in charge of each of them) that will be detailed in the following of this section. We refer to the case study of Section 4.5 for a more detailed description.

1. Analysis: definition of project activities (*manager*)
2. Modelling: definition of the SDs and the DD (*manager*)
3. Model annotation: specification of proper parameters and values (*manager*)
4. SPE models generation: derivation of the SM and MM models *(automatic*)
5. Model evaluation: resolution of the EQNM and derivation of the relevant predictions (*automatic*)
6. Analysis of results: evaluation of the results obtained (*manager*)

Figure 1 The Propean application steps

1. Manager: *Analysis*

In this step the project manager defines the project activities under consideration. In particular he/she has to associate with each activity an estimation of the time and resources necessary for completing it and the roles of the people involved[2]. In organizations with stable processes, this information can be derived from previous experience on similar projects.

2. Manager: *Modelling*

The results of analysis in Step 1 have to be modelled as RT-UML diagrams. In particular the manager should describe, in one or more SDs, the scenario(s) representing the adopted release process. In the SDs the objects represent the teams involved, and the messages represent the requests of execution of a set of activities or correspond to information/data exchanged between the teams. Moreover the manager

[2] Note that this is a classic manager duty and not a specific request of the proposed method.

should construct a Deployment Diagram (DD) modelling the resources available and their characteristics. In this case the nodes of the DD can be associated with: classic resources (device, processor, database), different project teams, communication means such as, for example, the intranet device. The links between the DD nodes represent the communications between the teams or the documents exchanged inside the organization. We note that the manager does not need to repeat this step from scratch each time he/she needs to make estimations about a project. In a mature organization, for similar products, the effort needed to derive this reference structure will be made only the first time. The same diagrams can then be re-used for subsequent similar applications, by possibly updating the associated parameters, as will be described in the next steps.

3. Manager: *Model annotation*

The two types of diagrams developed in Step 2 must be annotated with the proper values and parameters. The project manager should express, by using a comment-based annotation, the attributes associated with events and actions of the diagrams. In particular, referring to the attributes relative to the PA profile described in Section 4.3.1 the PA attributes of the closed workload will be associated with the initial action of the SD; those of the steps will be linked to each of the subsequent message activations; those of the resources will be related to each of the nodes of the Deployment Diagram. In Figure 4 and Figure 5 we report an example of the resulting SD and DD. The details of these figures will be described in the next section.

4. Automatic: *SPE models generation*

By applying the method proposed in [CM02], and described in Section 4.4.1, it is possible to derive a model for the planned activity (the SM based on EG) and a model for the involved teams and resources (the MM based on EQNM).

5. Automatic: *model evaluation*

The EQNM obtained in the previous step, which represents the teams and activities, can be solved to obtain relevant results such as: the predicted completion time for the project (or for a single phase), the resource utilization rate, the best resource distribution with respect to a given completion time, and so on.

6. Manager: *analysis of results*

The results automatically obtained in Step 5 are analysed by the project manager and, if different from those expected, he/she can go back to Step 1 (or 2), make some modifications to the diagrams or to the assigned parameters, and repeat the process until the desired results are obtained.

4.4.1 EG and EQNM Derivation

With reference to Step 4 of the Propean methodology described above, we explain here the process applied for automatically deriving the EG and the EQNM. In short it consists of the analysis of the SD and DD designed by the project manager as described in the following phases.

1. Analysis of the SD

In this phase we analyse the SDs separately and in particular the messages exchanged between objects considering their respective lifeline. In particular in each SD we associate with every interaction a tuple *(l, A_1, A_2, PApar)* where:

- *l* is the label of the SD interaction arrow,
- A_1 is the name of the object from which the arrow starts,
- A_2 is the name of object at which the arrow ends,
- *PApar* represents the performance annotation of <<PAstep>>

The SD is now translated into a high level EG (called meta-EG). Each node in the EG identifies an interaction, and corresponds to the set of operations performed in relation to that interaction. Every node in the meta-EG is labelled with the tuple *(l, A_1, A_2, PApar)* that characterizes the translated interaction. Figure 2 illustrates the skeleton of the algorithm used for the EG generation with proper labels for a very simple SD.

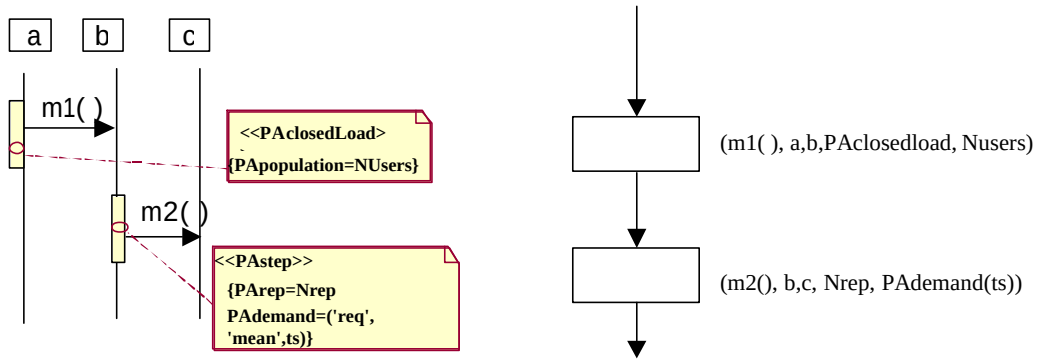


Figure 2 Labeled Meta-EG generation

Depending on the degree of detail the manager adopts, the SD may be too complex to allow the generation of a unique comprehensible software model (EG). In this case before generating the EG it would be convenient to check if there are parts

(operations in the SD) that can be grouped together. For example, we could aggregate a set of operations that is repeated many times in the SD, or that belongs to the same process phase. In this case a high-level EG is generated and expanded, when necessary, into sub- EG.

The Execution Graph (called meta-EG) obtained in this step includes only five types of nodes: basic, branching, cycle, fork and join [Appendix A, SM90, SW01]. Each basic node is labelled with the tuple *(l, A_1, A_2, PApar)* identifying an interaction, and corresponds to the set of operations that are carried out by component A_1 before interacting with A_2 through *(l, A_1, A_2, PApar)*. This set of operations is translated into an EG node and possibly connected to another node by a pending arrow. In case of multiple interactions, a fork node is placed before considering the sequence of the outgoing interactions. The multiplicity of the fork node determines the number of pending arrows associated with it and is obviously equal to the cardinality of the multiple interactions (i.e., the number of different threads originated by this interaction). Figure 6 and Figure 7 show respectively the high level EG obtained from the SD of Figure 4 and a sub-EG relative to the block problem analysis.

2. Analysis of the Deployment Diagram

The use of the information contained in the Deployment Diagram is twofold. On one hand, tailoring the meta-EG to the specific platform can be performed, thus obtaining an EG-instance; on the other hand an Extended Queueing Network Model (EQNM) can be obtained representing the hardware platform hosting the software system.

We start by describing the stepwise process adopted for deriving the EG-instance. In our case the nodes of the DD does not represent the hardware resources, but the different teams. The components inside a node represent the tasks that the team must perform (obviously, a team can be composed of one or more people). Project phases can be carried on with the collaboration of components living inside different nodes of the DD. Specifically the names of the interacting components within the meta-EG block labels are substituted with the names of the team that accomplishes the operation and the values of the relative *PAattribute*. Furthermore, when the names of the interacting components are different in the label, an overhead delay due to coordination among project teams (e.g., team meeting) is added to the performance model. In this way the node label in the EG-instance corresponds to the *demand vector*, which specifies for each team the work-demand relative to the modelled

operation. Figure 7 shows an example of an EG-instance, including the demand vector specification.

The EQNM topology can be derived in a straightforward way from the information collected in the DD. As already stated, in our case the service centres model the project teams involved in the software processes, so the number of service centres in the network correspond to the number of teams. The connections between different service centres are derived from the communications represented in the DD. The values of the attributes associated with the different nodes of the DD are used to better specify the characteristics of the EQNM service centres.

Subsequently by using well-known performance techniques [CM02, SM90, SW01], the obtained EG-instance is combined with the EQNM to achieve the complete definition of the queueing model, as in the traditional SPE approach. The obtained model is then solved by use of the classic solution technique and tools [LA83, SW01] to obtain the performance indices of interest.

4.4.2 Architecture of the Propean Tool

The final goal of this research is an automated environment that the manager can easily consult in his/her everyday activity to obtain advice for sound decision-making. With reference to the stepwise procedure described in Section 4.4, this environment should incorporate a tool fully automating steps 4 and 5 (the SPE related computations), and provide support to the other steps pertaining to the manager as well, facilitating the RT-UML modelling of the workflow and the resources according to the required formats.

Currently we have already available some small pieces of such a tool (which we used to process the case study presented here), while further implementation work is ongoing to complete the platform. In this section we overview the architecture of the Propean tool. To make the SPE calculations, Propean transforms the UML model annotated with performance information into an Extended Queueing Network performance model [LA83]. Following [UMLP] the input to our transformation algorithm is a file containing an annotated UML model translated in XML format according to the standard XMI interface [UMLP], and the output is the corresponding EQNM model description file, which can be read directly by existing EQNM solvers [LA83]. The tool architecture is illustrated in Figure 3. A UML tool (such as Poseidon [POS]) processes the input diagrams and generates the XML file.

The UML to EQNM transformation component takes as input the XML file and produces as an output a file describing the performance model.

Specifically, the EQNM model structure is generated from the high-level software architecture described in the DD indicating the allocation of software components (in our cases the activities) to hardware devices (in our case the teams). The EQNM model parameters are obtained from detailed models of key performance scenarios, represented in the SDs. The derived performance model goes to a performance model solver (EQNM analytical solver or simulator) which derives the performance analysis results. Finally the Results Convertor analyses the performance results and convert them back in the UML model as constraints on some PAattribute, thus completing a round-trip tour.

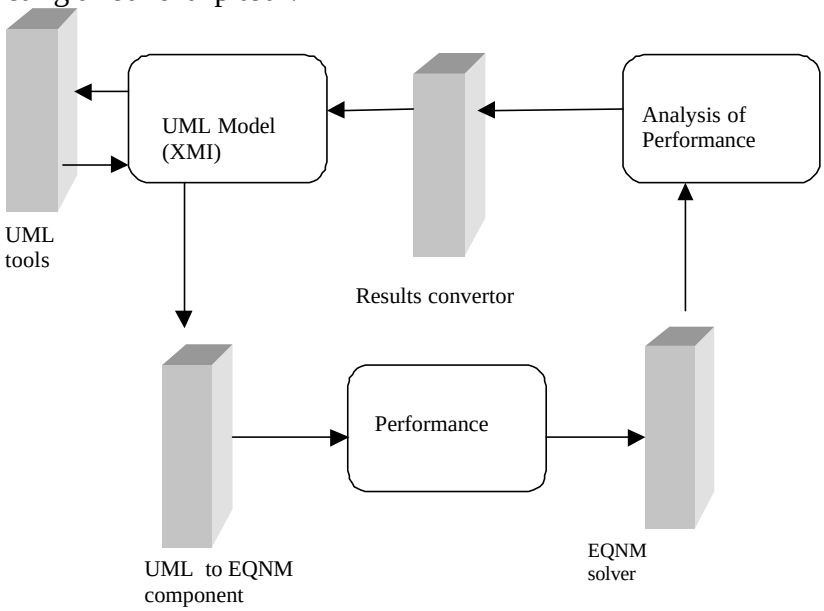


Figure 3 Propean Tool Architecture

The policy we are pursuing in the implementation is that, where available, we use existing tools, and integrate them into the Propean tool. For instance, we obviously do not want to develop a new UML tool. In this regard, we notice that the release of the RT-UML profile has occurred quite recently and there are not yet commercial tools specifically handling it or supporting the Performance profile.

Therefore, the XML files must be processed to “attach” the tagged values associated to different model elements with the stereotypes.

4.5 Propean for Managing the Testing Phase

In every process development, as best practice before facing any of the development phases is to establish an accurate and truthful Development Plan in which the required resources are assigned and the people scheduled. Considering in particular the testing phase, Propean methodology, by applying the well-known techniques from the field of computer performance engineering, can facilitate the definition of the Test Development, providing the managers with sound and reliable solutions. Performance analysis techniques are used to predict the outcomes that will result from manager's assumptions and to figure early out whether under the current workflow the settled objectives will be met. In particular, Propean provides managers with both the resources distribution (i.e. people assignment) and the prediction about the completion time of the processes under development. Specifically, as stated in the preface the Propean as support for the manager's decisional process in multiproject management can be useful at any stage of development, as soon as the project manager is called to dynamically make the most appropriate decision based on the actual project status and the emerging circumstances. We will show an examples of this in Section 4.6.

4.5.1 Case Study

We investigate here the release decision for a software product driven by product quality, measured in terms of bugs found. More precisely, we suppose that, as usual, the testers report each failure found during the test execution in a form, called the trouble report, and that the product will be released only after the testing is completed with no trouble report left open.

We consider that at the beginning of the test phase the manager faces either of two different situations:

i) considering the actual personnel availability, he/she wants to early predict the expected time until release;
ii) for a fixed release time, previously established on the basis of customer exigencies, he/she wants to decide the most adequate personnel configuration to respect the time constraints.

In this section we illustrate the use of Propean for pursuing either goal (i) or (ii) considering a simplified case study derived from [KFN99], to which we refer for further detail.

As a first step, we model the organization structure of the company considering the testing stage and the management of reported problems (we disregard the teams not directly involved in these activities). The organization is composed of a project manager PM, a test team T (1÷3 people), a development team D (2÷4 people) and the system architects SA (1÷2 people).

The testers begin to execute the planned test cases and every few days (we assume 3 in this example), they insert the trouble reports in an on-line database, called the tracking system TS, which only the testers and the project manager can modify.

At each TS update, the PM analyses the trouble reports and seek the proper solutions for each reported problem. We consider three possible outcomes from his/her analysis:

- The problem must be fixed: the PM classifies the problem as "open" and passes it on to the developers. In this case study for simplicity we assume no prioritisation politics among failures, i.e., all reported problems are assigned the same severity (different priorities could also be handled, but the example would be more complicated).
- The problem can be deferred. The PM chooses to leave the problem in the current version of the product and to fix it in a subsequent release. The problem is classified as "deferred".
- The problem is not recognized as such. From the trouble report analysis the PM concludes that it is not a real problem, because the program was actually supposed to work in that way. The problem is classified as "as designed".

The TS update with the problem classification as "deferred" or "as designed" by the PM closes the trouble report (at least for this product release). If instead the problem is classified as "open", further actions must be taken as described below.

On receiving the open problem reports from the PM, the developers first analyse them to check whether they have enough information to fix the problems or need further explanation from the testers about the failure symptoms. In the latter case, the workflow may include an interaction cycle with the testers. Occasionally, the developers may realize that the fix requires a major design change and so inform the PM, who may require the intervention of the software architects to modify the design, after which the developers modify the code accordingly.

After every problem fix, the testers have to retest the modified parts of the program (regression test). Thus they either classify the problem as "closed",

consequently updating the associated trouble report in the TS, or possibly generate further trouble reports containing the new problems found during the test phase.

Given this abstract workflow of the activities and personnel involved, the project manager can periodically analyse the status of the TS in order to:

Case i) estimate the expected time at which the product can be released, that is when the TS only contains problem reports classified as “deferred” or “as designed”, i.e., there are no remaining “open” problems;

Case ii) derive the most efficient personnel organization for releasing the product within the established time constraints.

In Case i), if the estimated release time is too late, for example with respect to market demands, the PM has to take the proper corrective actions. For instance, the PM could increase the number of people involved in the development or test phase or else decide to pursue a later release date. Alternatively, if the involved personnel are handling several projects simultaneously, the PM could decide to temporarily divert the people from one or more of the concurrent projects to focus on this one. Similar considerations can be made for Case ii). In both situations, it is very important that the PM base his/her resolution on a reliable estimate, not on a subjective guess, and that he/she can objectively take into account all the likely combinations of events.

This is the purpose of the methodology presented in the following section: we intend to supply the project manager with a tool that uses performance engineering techniques to:

- Predict the release time, also allowing for multiproject management, i.e., the teams are not dedicated full-time to a single project
- Identify (by looking at the personnel rate of utilization) the component that represents the bottleneck and is mainly responsible for the release time delay
- Identify the most convenient team composition in order to ensure that all the projects are released within the deadline agreed upon with the customer, or within the budget allowance.

4.5.2 Details of the Methodology

In this section we describe the application to a case study of the Propean methodology (Section 4.4).

1. Analysis: In this case the project manager decides to focus analysis on the testing phase. During this step he/she has mainly to define the boundary conditions, such as for example the resources involved and the strategy to adopt for project

release, and to establish which parameters (symbolic expressions) are relevant for the estimation, possibly postponing their evaluation to Steps 5 and 6.

2. Modelling: during this step the manager develops the SD and the DD representing respectively the sequence of the activities performed during the testing phase and the overall organization of the different teams. These two diagrams are presented respectively in Figure 4 and Figure 5; the meaning of the stereotypes and tagged values of RT-UML will be explained in the next step.

3. Model annotation: The SD and the DD developed must be annotated with the proper parameters.

Considering the SD, the attributes relative to the closed workload are associated with the first action of this diagram. The note must report the name of the stereotype (<<PAClosedLoad>>) followed by the parameters associated with the PA attributes which are:

- PApopulation = *$Nuser* that represents the number of jobs in the scenario: in our context, the number of projects contemporarily under testing. The symbol *$* indicates that *$Nuser* is a variable that the project manager will instantiate with an appropriate value before starting the automatic derivation of the required estimations.
- PAresptime = *('msr','mean', $t_to_release)* that represents the completion time and is one of the expected results. It is modelled as a measured (*'msr'*) distribution whose mean is expressed by the variable *$t_to_release*.

The other steps of the SD are annotated with the stereotype <<PAstep>> associated with different PA attributes, depending on the activity considered.

Considering the second and third step of the SD, every three days (*Nrep*) the testers insert in the database a certain number of trouble reports (denoted as *$N*). The insertion has a mean value equal to *ts*. In other words, *$N* and *ts* are the values to be estimated by the project manager, possibly with the help of testers. They are associated with the PA attributes as follows:

- PArep = *Nrep* number of insertions in the data base
- PAdemand = *('req','mean', ts)* the execution demand of this step on its host resource, i.e., the time necessary (*'req'*) for the testers to insert a trouble report in the database, follows a distribution whose mean is given by *ts*.

When the Project Manager analyses the trouble reports, he/she observes a variable number of bugs reported (*$N*). The value to be associated with this variable generally depends on the project typology and can be estimated for a family of

similar products. The Manager can classify each bug as “open”, “deferred” or “as designed” with a given probability (denoted as *p_fix, p_def, p_des,* respectively) and spending a certain amount of his/her time (denoted as *t_fix_PM, t_def_PM, t_des_PM*, respectively). The Project manager must estimate the associated values based on his/her experience. For example let us considering the deferred bugs: the value *$N*p_def* will give the number of bugs classified as deferred among the *$N* reported and *t_def_PM*$N*p_def* will represent the time necessary for the project manager to deal with them. The situation described is simply modelled by associating with the relative step the stereotype <<PAstep>> with attribute:

- PAdemand = *(‘req’,’mean’, k)* where *k* can assume values *t_fix_PM*$N*p_fix*, *t_def_PM*$N*p_def, t_des_PM*$N*p_des* depending on the considered SD step. It represents the time necessary (‘*req’*) for the project manager to deal with the different trouble reports and follows a distribution whose mean is given by *k*.

Similar consideration can be also done for the other steps of the SD, annotating each step with the parameters of interest and estimating the required values.

Considering the DD, its nodes can refer to both classical resources (device, processor, database) and people teams. Moreover, the DD also models the communication nodes: for instance, the Intranet to access the database TS and a meeting room symbolizing a “communication channel” between different teams. Each node represents a kind of resource and therefore it is necessary to associate with each resource a stereotype <<PAhost>>. Then, depending on the resource considered, the associated PA attributes are:

- PAschdPolicy=*P* where *P* can be equal to FIFO, PS or PR and models the strategy by which the resource handles the different jobs.
- PApreemptable= *Yes* In the case study it is supposed that only the System Architect team can be interrupted during his/her work.
- PAutilization=*$Util* represents the rate of utilization of the different resources. The value associated with this variable is an analysis result.
- PArate=1 the resource works full-time on the assigned job.

Pathroughput=*Np* represents the amount of work provided per unit of time (a day in the case study) by a person belonging to a certain team. This value is normalized to 1 in case of a single person, 2 when two people work together, and so on.

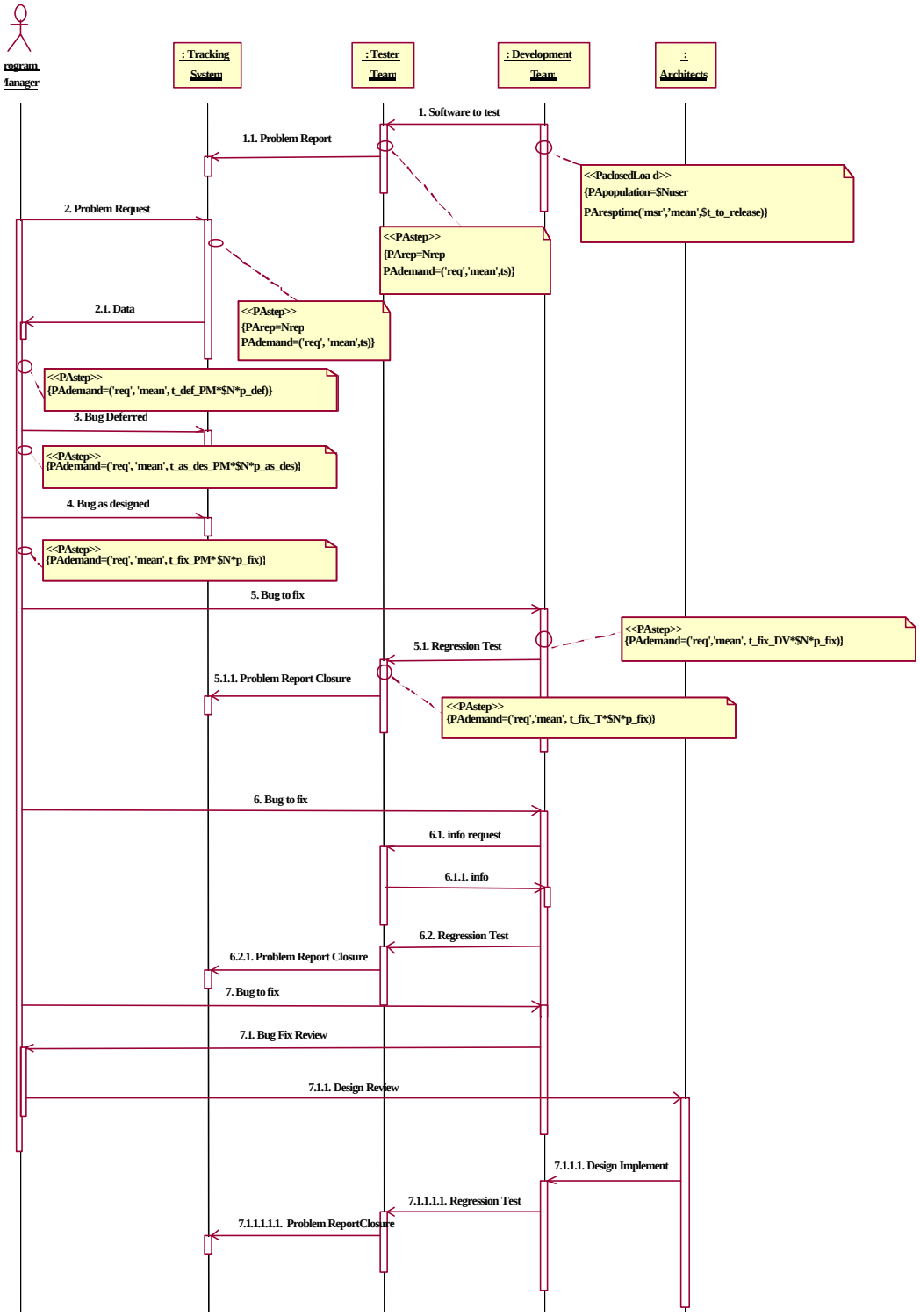


Figure 4 Sequence Diagram

4. SPE model generation: Following the steps described in the previous section, the corresponding EG and EQNM can be automatically derived from the annotated SD and DD. Figure 6 and Figure 7 illustrate the high level EG and one of the low level EGs obtained from Figure 4 and Figure 5 respectively, while Figure 8 shows the EQNM with a team composition made of: 1 project manager, 1 software architect, 1 tester and 2 developers (1PM, 1SA, 1T, 2D).

With respect to the SD and the DD, in this step we have made the following choices:

i. The database TS and the connected Intranet have not been modelled, because the times involved in the TS accesses are orders of magnitude less than the times required by the activity steps (msecs vs. days);

ii. The meeting room has been introduced as a delay centre modelling the communications with the manager.

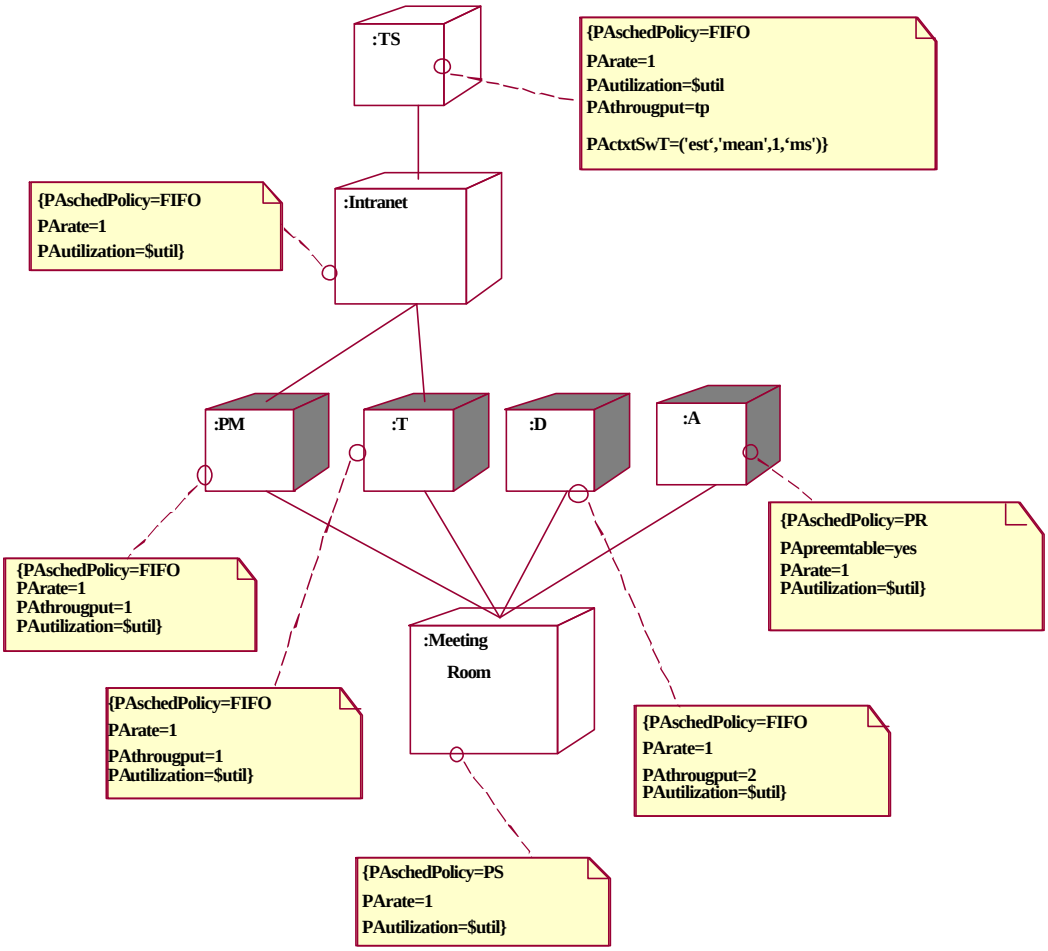


Figure 5 Deployment Diagram

Figure 6 represents an EG at a high level of abstraction modelling the main activities of the testing phase without details, while Figure 7 shows the details of the block named “problem analysis”, by illustrating several activities modelled in Figure 4. Moreover, the demand vector for each block is derived by combining information coming from annotations in the SD and in the DD.

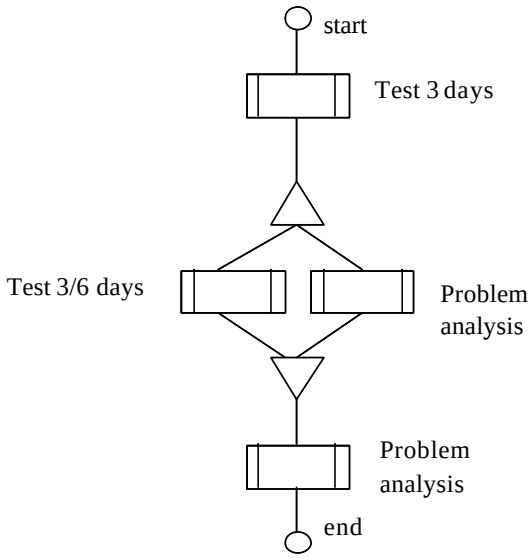


Figure 6 High level EG obtained from SD in Figure 4

For example, the first block is called “3+4” because it models the interactions 3 and 4 in the SD; its associated demand vector represents the service demand to the resources involved in the scenario for the management of bugs that are deferred or classified “as designed”. In such a case only the manager is involved and his/her service demand can be derived from the annotated SD as

(t_def_PM $N*p_def + t_as_des_PM*$N*p_as_des).*

Note that the different kind of projects (depending, for example, on the test duration or on the number of bugs) generate different instances of the demand vectors for the EGs in Figure 6 and Figure 7, and therefore different routing chains in the EQNM. The possible choices lead generating different models to be evaluated in the next step.

5. Model evaluation: several analyses can be done by assigning different values to parameters in the EQNM. Examples of various model evaluations are illustrated in the next section.

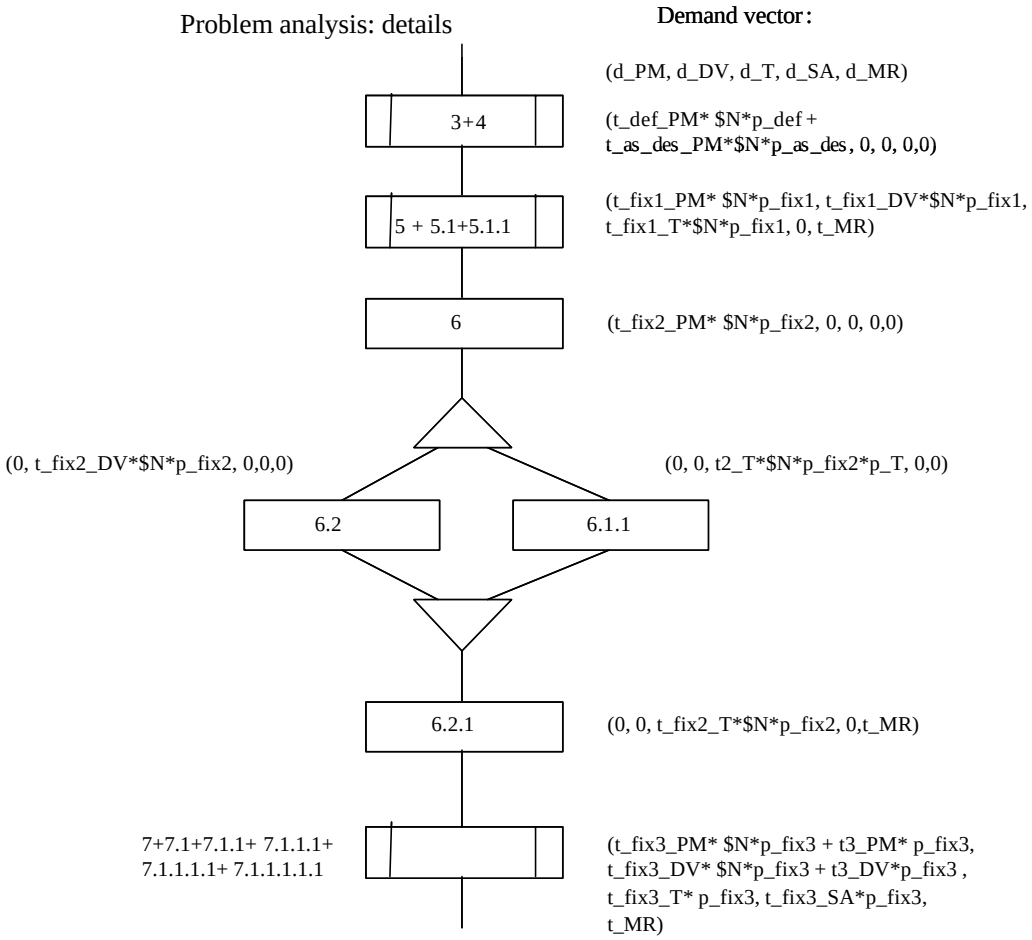


Figure 7 The low level EG for "problem analysis" block obtained from SD in Figure 4

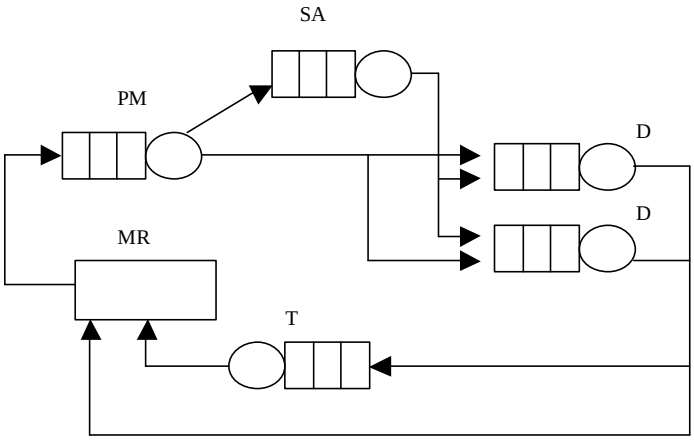


Figure 8 The EQN Model obtained from SD in Figure 4 and from DD in Figure 5

6. Analysis of results: the manager can make several kinds of decisions by analysing the results obtained in the previous step. Again, an example of this analysis is illustrated in the next section.

4.5.3 Discussion and Results

It is important to point out that on the manager's side the effort required to employ the methodology is to explicitly derive in a SD (such as the one shown in Figure 4) a high level model of the activity workflow and in a DD (as in Figure 5) the organization structure. He/she does not need to know all the other details on how such models are then translated into SPE models and then solved.

We understand that even the derivation of the RT-UML diagrams could be felt at first impact to be an undesirable extra burden for the already overloaded manager. However, objectively it should not take much effort: if one has a clear view (as plausibly the manager must have) of how the development process is structured and which activities are to be accomplished, and their mutual influences, deriving the RT-UML diagrams that depict them at a high level of detail should not take much labour, especially with the support of an appropriate interactive tool. Besides, we expect that the returns make it worthwhile.

In fact, once such diagrams have been derived, various interesting analyses can be conducted in completely automated way. The manager can make different assumptions on the parameters of the modelled scenarios and automatically obtain a reliable prediction of what will be the outcomes consequent to each assumption.

We present in the following two orthogonal applications of Propean to the described case study, illustrating for each of them the parameters to be introduced and the kind of estimations that can be derived. We consider the two situations in which: (i) the manager wants to predict the completion time of the testing phase (Section 4.5.3.1); or (ii) he/she wants to derive the most efficient personnel distribution for completing the testing phase within a fixed time deadline (Section 4.5.3.2).

Generally for each diagram several parameters can be varied, depending on the desired prediction. We have considered the following parameters: the estimated duration of the test period, the number of registered trouble reports, the composition of the involved teams, and whether they are fully dedicated to the examined project or instead are contemporaneously handling other projects.

4.5.3.1 Estimating the Completion Time

Considering the case study described in Section 4.5.1, we illustrate some plausible situations: in Propean, each different situation corresponds to a variation in the parameter values in SD and DD. For example a possible situation could be represented by the following assumptions:

- The planned duration for the test phase of a given product is six days (considering the attribute PArep in the second step of the SD, the parameter *Nrep* is set equal to 2);
- For the type of project under test, based on his/her experience, the manager assumes that the number of trouble reports issued will be equal to 10 (considering the attribute PAdemand in the third step of the SD, the parameter *$N* is set equal to 10)
- The personnel in charge for the test and debug phase consists of one tester, two developers and one software architect (plus of course the manager): this configuration is denoted as 1PM, 1SA, 1T, 2D; (as illustrated in the considered DD)
- The tester and the two developers are in parallel engaged in another project (the parameter *$Nuser* of the SD is initialised to 1).

For each parameter configuration, using Propean the manager can thus obtain, early in advance of the testing phase, a prediction of the time in which the product will be ready for release (i.e., no more open trouble reports exist).

		Planned Test Duration=3 days				**Planned Test Duration=6 days**				**Planned Test Duration=9 days**			
	Project #Bugs	**T&D Full Ded**	**D Shared 1**	**T&D Shared 1**	**T&D Shared 3**	**T&D Full Ded**	**D Shared 1**	**T&D Shared 1**	**T&D Shared 3**	**T&D Full Ded**	**D Shared 1**	**T&D Shared 1**	**T&D Shared 3**
1PM 1SA 1T 2D	2	5	6	8	13	8	10	14	25	11	13	17	28
	10	9	10	11	17	12	13	17	29	14	16	20	31
	20	14	15	16	22	16	18	22	32	18	20	24	35

Table 1 Estimated time to release in days

In Table 1 we report the results obtained for different parameter values. In the table the estimated time to release is measured in days, considering one working day

to be equal to 8 hours (optimistic bound), and the results are rounded to the closest integer. The table shows the release time when the planned duration for the test and debug phase is three, six or nine days (with a group of four columns for each case), and when 2, 10, or 20 trouble reports are issued, as indicated in each row.

For each case, then, we derive the estimate when the resources (the people) are fully dedicated to the project under exam (denoted as T&D Full Ded); the test team is fully dedicated, while the developers are handling this and another project (D Shared 1); both the testers and the developers are handling this and another project (T&D Shared 1), and finally both the testers and the developers are handling three more projects in addition to this one (T&D Shared 3).

Going back to the situation described above, in which the test duration was assumed equal to six days and the number of trouble reports to 10, the time necessary for completing the testing phase is estimated to be 17 days from the start of the test phase (2^{nd} row, 7^{th} column). If the manager was pointing towards a much earlier release deadline, Propean shows it is unlikely that he/she will be able to meet it. Let us assume that the target release deadline is 12 days. Even considering the more optimistic hypothesis that only two bugs are encountered, in the present configuration the release time would be not shorter than 14 days (1^{st} row, 7^{th} column). Thus, either the manager can accept a more relaxed deadline, or he/she takes some countermeasure. In particular, if the project under exam has high priority, a possible solution could be to take away from the other parallel project the resources (the tester and the developers) that are necessary to complete this one. In this case if they are fully assigned to the completion of the testing phase of this project, then the predicted release time with 10 bugs is reduced to 12 days (2^{nd} row, 5^{th} column), which was the target deadline. Thus, by means of simple SPE analyses, the manager gets statistical predictions that can support his/her decisional process.

On the other hand, adopting the latter solution results in an increased project cost, due to the under-utilization of certain personnel categories. Another relevant parameter the manager should consider before taking any decision is in fact the rate of utilization of the teams involved. This analysis is automatically obtained with the parameter assignments used for the estimation of completion time and can be very useful not only to better administrate human resources, but also to identify the bottlenecks when a phase takes too long.

		Planned Test Duration=3 days								Planned Test Duration=6 days								Planned Test Duration=9 days							
	Project #Bugs	T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3		T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3		T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3	
		T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D
1PM 1SA 1T 2D	**2**	21	23	15	59	99	55	100	77	66	10	50	56	99	53	100	76	79	70	65	54	99	52	100	75
	10	29	29	25	57	99	56	100	74	56	29	46	58	99	55	100	76	69	16	59	56	99	54	100	76
	20	31	31	28	55	99	55	100	73	49	30	46	56	99	55	100	74	69	23	56	55	99	55	100	76

Table 2 Percentage of utilization rate of testers and developers

		Planned Test Duration=3 days				Planned Test Duration=6 days				Planned Test Duration=9 days			
	Project #Bugs	T&D Full Ded	D Shared 1	T&D Shared 1	T&D Shared 3	T&D Full Ded	D Shared 1	T&D Shared 1	T&D Shared 3	T&D Full Ded	D Shared 1	T&D Shared 1	T&D Shared 3
1PM 1SA 2T 2D	**2**	5	6	6	10	8	10	12	19	11	13	14	22
	10	9	10	11	14	11	13	15	22	14	15	17	24
	20	14	16	16	20	16	17	20	27	18	19	21	28

Table 3 Estimated time to release in days

		Planned Test Duration=3 days								Planned Test Duration=6 days								Planned Test Duration=9 days							
	Project #Bugs	T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3		T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3		T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3	
		T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D	T	D
1PM 1SA 2T 2D	**2**	12	24	7	59	55	58	76	77	37	10	26	29	55	55	74	75	40	70	34	54	55	54	70	76
	10	14	29	13	57	55	56	76	75	30	22	24	31	56	56	75	75	36	17	33	56	56	55	72	76
	20	15	30	14	55	54	55	75	70	26	27	22	31	56	57	74	74	33	24	28	55	56	55	74	74

Table 4 Utilization rate of testers and developers

In Table 2 we report the percentage of the utilization rate, denoted by ρ, for the tester and the developers considering the same parameter assignment in Table 1. This index is measured by the ratio between the frequency at which requests arrive, and the frequency at which the processing element (in our case a team) can deliver services. The utilization rate varies between 0 and 1, where 1 means that the resource

is saturated, and can represent a bottleneck; 0 means it is idle, and a good utilization is somewhere in the middle.

In the initial configuration we assumed one tester and two developers, employed in this and in another project. We can see that the bottleneck is clearly the tester, as the utilization rate percentage is computed to be 99%, while the two developers are well employed, with a rate of 55%. Deciding to fully dedicate one tester and two developers to the test and debug phase allows the manager to meet the deadline, but in such a configuration the developers are under-utilized, at 29%.

One further possibility to explore could be to devote one tester at full time, while leaving the two developers on both projects. In such a configuration we would get a release period of 13 days, but the resources are better employed (the tester 46% and the two developers 58%).

Analysing the results in Table 1, another interesting fact can be observed: although obviously the duration of the test and debug process can be greatly influenced by the number of bugs found, a rational organization of the personnel is more crucial, especially for large enterprises dealing with several development processes in parallel. The release delay increases faster as the teams get involved in more simultaneous projects than if we increase the estimated number of bugs.

For instance, considering a large product with a planned test period of 9 days, when all the resources are fully dedicated the expected release time, even foreseeing 20 bugs, is 18 days, against the 20 days estimated to handle half (i.e., 10) bugs if the tester and the developers are contemporaneously employed in another project. If we further consider a configuration in which the tester and the developers are handling three more projects, even though in this project we optimistically assume finding only two bugs, handling them would take 28 days.

Another possible countermeasure when the predicted release time exceeds the target deadline is to add more personnel to the development of product. Using Propean, revising the estimates is immediate and again it only consists of changing some of the configuration parameters.

Let us consider, for example, that the personnel in charge of the test and debug phase consists of two testers, two developers and one software architect, plus of course the PM. This configuration is denoted as PM1, SA1, T2, and D2. We report, in Table 3 and Table 4 respectively, the estimated time to release and the utilization rate of the testers and developers.

Considering the initial situation in which the test duration was equal to six days, the number of trouble reports to 10 and the committed release time 12 days, even if one more tester is added, the product would be ready in 15 days (as reported in Table 3, 2nd row, 7th column), instead of 17 as with the previous configuration, but this may not be sufficient. In this new configuration the manager would be able to meet the target deadline only if the estimated number of bugs is two.

The utilization rate of testers with ten bugs (Table 4) is equal to 55% (instead of 99% of the previous cases). This means that personnel organization in this case is better than before and the tester resource is no longer the bottleneck of the development process. As shown in the table, in this case assigning the testers and the developers full-time to the project, or even only the developers, would be meaningless. Their utilization rates in the two cases, (30% and 22%) and (24% and 31%) respectively, reflect an under-utilization of the resources.

4.5.3.2 Deriving the Best Personnel Distribution

As in the previous section, we discuss several situations for illustration purposes. Considering the case study, in this section we report the results obtained under the following assumptions:

- The planned duration for the test phase of a given product is 3 days, (considering the attributes PArep in the second step of the SD the parameter *Nrep* is set equal to one);
- For the type of project under test the manager assumes that the number of trouble reports issued will be equal to two (considering the attributes PAdemand in the third step of the SD the parameter *$N* is set equal to two);
- The personnel in charge of the test and debug phase consists of one software architect, the program manager, while the number of testers and developers is variable. The configuration is denoted as 1PM, 1SA, T*t*, D*d* where the variables *t* and *d* indicate the values to be established by using Propean.

More precisely, the goal of the manager, given the above parameter assignment, is to define the values for the variables *t* and *d*, i.e. the best personnel assignment, so that the project can be released within no more than seven days (considering one working day equal to 8 hours, and the results rounded to the closest integer).

Table 5 reports some of the results obtained when the planned duration for the test and debug phase is three days, and considering the testers and developers fully dedicated to the project under exam (denoted as T&D Full Ded); the testers fully

dedicated, while the developers are handling this and another project (D Shared 1); both the testers and the developers are handling this and another project (T&D Shared 1), and finally both the testers and the developers are handling three more projects in addition to this one (T&D Shared 3). In particular we suppose that when a team (resource) receives the request for a job, the task is performed by only one of the people available at that moment.

Configuration 1PM, 1SA, T*t*, D*d*		**Planned Test Duration=3 days**			
		T&D Full Ded	**D Shared 1**	**T&D Shared 1**	**T&D Shared 3**
#Bug 2	***t=1, d=2***	5	6	8	13
	t=2, d=2	5	6	7	10
	t=3, d=4	5	5	6	8
#Bug 10	***t=1, d=2***	9	10	11	17
	t=2, d=2	9	10	11	14
	t=3, d=4	9	9	10	12
#Bug 20	***t=1, d=2***	14	15	16	22
	t=2, d=2	14	15	16	20
	t=3, d=4	14	15	15	17

Table 5 Estimated completion time at various configurations

Configuration 1PM, 1SA, T*t*, D*d*		**Planned Test Duration=3 days**							
		T&D Full Ded		**D Shared 1**		**T&D Shared 1**		**T&D Shared 3**	
		T	D	T	D	T	D	T	D
#Bug 2	***t=1, d=2***	24	24	15	60	99	56	100	76
	t=2, d=2	12	24	0.7	60	55	58	76	77
	t=3, d=4	0.7	12	0.6	33	38	30	60	52
#Bug 10	***t=1, d=2***	30	29	25	57	99	55	100	74
	t=2, d=2	15	29	12	57	55	56	76	75
	t=3, d=4	10	15	0.9	30	38	30	60	50
#Bug 20	***t=1, d=2***	31	30	27	55	99	55	100	75
	t=2, d=2	16	30	14	54	55	56	76	76
	t=3, d=4	10	15	0.9	30	38	30	61	50

Table 6 Utilization rate (%) of testers and developers

Therefore, if the estimated number of trouble reports is equal to 2 and the target release date is 7 days, from the analysis of Table 5, a good configuration could be one tester and two developers completely dedicated to this project, one tester and two developers with developers handling this and another projects, or, alternatively, two

testers and two developers engaged at the same time in another project. The other configurations can be immediately excluded because beyond the deadline.

The manager can also derive a more precise evaluation of the cost of project realization, by using the utilization rate of the people involved. For this reason in Table 6 we report the corresponding percentage of the utilization rate of the tester and developers, while a more thorough discussion of the project cost is deferred to Section 4.5.3.3.

An alternative situation is that when a team (resource) receives the request of a job, the task is performed by all the people available in that moment. This means that if two people are available, they will work in parallel to complete the job. Table 7 and Table 8 report the estimated completion time for the different configurations and the percentage of utilization rate of the testers and developers. The different policy of job completion is incorporated by the resulting utilization rates of the testers and developers.

Configuration 1PM, 1SA, T*t*, D*d*		**Planned Test Duration=3 days**			
		T&D Full Ded	**D Shared 1**	**T&D Shared 1**	**T&D Shared 3**
#Bug 2	***t=1, d=2***	5	7	8	15
	t=2, d=2	4	6	8	14
	t=3, d=4	4	5	6	8
#Bug 10	***t=1, d=2***	7	10	11	17
	t=2, d=2	7	9	11	16
	t=3, d=4	6	7	8	10
#Bug 20	***t=1, d=2***	12	14	15	21
	t=2, d=2	11	12	13	19
	t=3, d=4	9	10	10	13

Table 7 Estimated completion time at various configurations

4.5.3.3 Cost Estimation

Achieving the target deadline is certainly important. However, another relevant factor that the manager must also consider is the associated cost. For each selected configuration we can derive a rough estimation of cost denoted as $C_G(i)$, i=1, …, N_c (total number of configurations), and computed as follows:

$C_G(i) = d(i) * \sum_{k \in \{T,D\}} [(p_k(i) * c_k)/s_k(i)]$

Where : d (i) denotes the working days for configuration i

$p_k(i)$ is the number of people in team k

c_k is the cost associated with each person in team k

$s_k(i)$ is the number of projects shared by team k in configuration i

Configuration 1PM, 1SA, T*t*, D*d*		Planned Test Duration=3 days							
		T&D Full Ded		D Shared 1		T&D Shared 1		T&D Shared 3	
		T	D	T	D	T	D	T	D
#Bug 2	***t=1, d=2***	26	28	13	59	99	56	100	77
	t=2, d=2	14	29	60	60	52	57	76	77
	t=3, d=4	11	18	60	35	38	32	61	53
#Bug 10	***t=1, d=2***	34	35	26	62	99	60	100	78
	t=2, d=2	19	38	13	64	56	60	77	78
	t=3, d=4	16	25	14	40	42	38	65	56
#Bug 20	***t=1, d=2***	36	37	31	62	99	61	100	78
	t=2, d=2	20	40	17	63	56	62	77	78
	t=3, d=4	17	26	16	40	42	39	63	54

Table 8 Utilization rate of testers and developers

For example let us compare, the configurations *a* and *b* $(a,b \in \{1,\ldots, N_c\})$, where

a corresponds to: 6 days of planned test duration, 10 bugs, T&D shared1, 1PM, 1SA, 1T and 2D

b corresponds to: 6 days of planned test duration, 10 bugs, T&D shared1, 1PM, 1SA, 2T and 2D.

$C_G(a) = d(a) * \sum_{k \in \{T,D\}} [(p_k(a) * c_k)/s_k(a)] = 17 [(1*c_T)/2 + (2*c_D)/2]$

$C_G(b) = d(b) * \sum_{k \in \{T,D\}} [(p_k(b) * c_k)/s_k(b)] = 15 [(2*c_T)/2 + (2*c_D)/2]$

Let us suppose, for the sake of simplicity that $c_T = c_D = c$, we obtain:

$C_G(a) = 17*c*3/2 = 25.5*c$

$C_G(b) = 15 *c*2 = 30 *c$

Therefore the manager should decide by comparing configuration *a* and *b*, whether the increase in the project cost is justified by a reduction by only two days of the completion time.

Other information that can aid the manager in decision-making is the computation, based on utilization rate, of a so-called “waste” factor, which is the cost of people under-utilization, as follows:

$W_G(i) = d(i) * \sum_{k \in \{T,D\}} [(p_k(i) * c_k*(1-\rho_k))/s_k(a)]$

Therefore for configuration *a* and *b* above, we can compute

$W_G(a) = 17 * [(1 * c*(0.01))+(2*c*0.45)/2]=17*c*0.46=7.82*c$

$W_G(b) = 15 * [(2 * c*(0.44)/2)+(2*c*0.44)/2]=15*c*0.88=13.2*c$

Note that, by use of the attribute PArate we can also give an idea of the expertise of people belonging to the team, with PArate $\in [0,1]$, where 1 denotes a highly skilled person, while 0 denotes a beginner. So, we can weight the cost c_k by use of PArate to take into account the cost of different people. Obviously, the other parameters such as the centre service time modelling people work also should be weighted by the PArate value. In the examples above we have supposed, for simplicity, PArate equal to one.

Note that currently, due to restrictions imposed by the PA profile, it is necessary to suppose that all people in the same team have the same skill and cost (an average value can be taken, if this is not the case).

4.6 Propean Applied to RUP

The Propean methodology is not limited to the testing phase, but can be also adopted for managing the other development phases as well as the entire life cycle as described in [BBM02a, BBM03]. In this section we present a more general case study encompassing the modelling of the whole Rational Unified Process (RUP) (see Chapter 3), which is one of the emerging processes adopted in the industrial context. In particular exploiting the RUP peculiarity of letting its regular updated exactly as the software products, [KR00], our intent is to augment RUP with the capability of producing reliable schedule and resource utilization estimates useful to RUP decision makers [BLM02].

If the goal of RUP is to produce high-quality software within predictable schedule and budget, we need a means of reliably drawing such predictions: for instance, how long will RUP take to process a certain project? How will RUP utilize the available resources? How is the RUP schedule affected by the concurrent processing of several projects? In this section we answer such questions on rigorous grounds, equated RUP to a product, of which we analyse the performance by applying the Propean methodology, just as we do with any other critical product.

As stated in Section 4.4 for this purpose it is necessary first of all to derive the RT-UML diagrams that model the RUP process as configured to fit the needs of the specific organization under exam. This phase pertains to the manager, who has to model the process and the workflows to be instantiated. This may seem to be a heavy

requirement; in practice, the UML process model can be derived once for an organization, and then at each new application of the technique to a specific project, the manager only needs to update the parameters in the diagrams (such as the number of people involved, and the estimated duration of the single activities). The derived diagrams are then processed to obtain the performance estimations.

In the next section we discuss the steps necessary for applying the Propean methodology while in Section 4.6.2 we present the case study provided by Ericsson Lab Italy (ERI). Finally in Section 4.6.3 we present some results and discussions.

4.6.1 Details of the Methodology

In this specific case the Propean application to RUP is divided into two steps. The first, called *RUP modelling* (Step 4.6.1.1), consists of the description of the functionality and architecture of the RUP product by means of UML diagrams appropriately annotated according to the RT-UML profile. The second step, called *RUP customisation* (Step 4.6.1.2), represents the core of the Propean application. The UML diagrams developed in the previous step are refined and completed by the manager, accordingly to personnel availability and process exigencies. In the next subsections the main aspects of the two steps will be briefly described.

4.6.1.1 RUP Modelling

In this section we present a brief description of the procedure we used to represent RUP applying Propean. As for the development of any other software product, the first step is to describe the system functionalities (i.e. the process activities) in terms of Use Cases (UCs). This description follows an iterative process, incrementing at each iteration the level of detail of the system functional specification. We begin therefore by representing the interaction of the external actors (in our case, End User, Customer and Stakeholder) first with the different RUP phases (Inception, Elaboration, Construction and Transition), and then with the single workflows. The description is further refined representing the interaction of the external and internal actors (one actor specification per role) with the workflow activities in the different phases. Finally for each activity an annotated Sequence Diagram (SD) representing the roles interaction is developed. Figure 9 reports an example of the SD used.

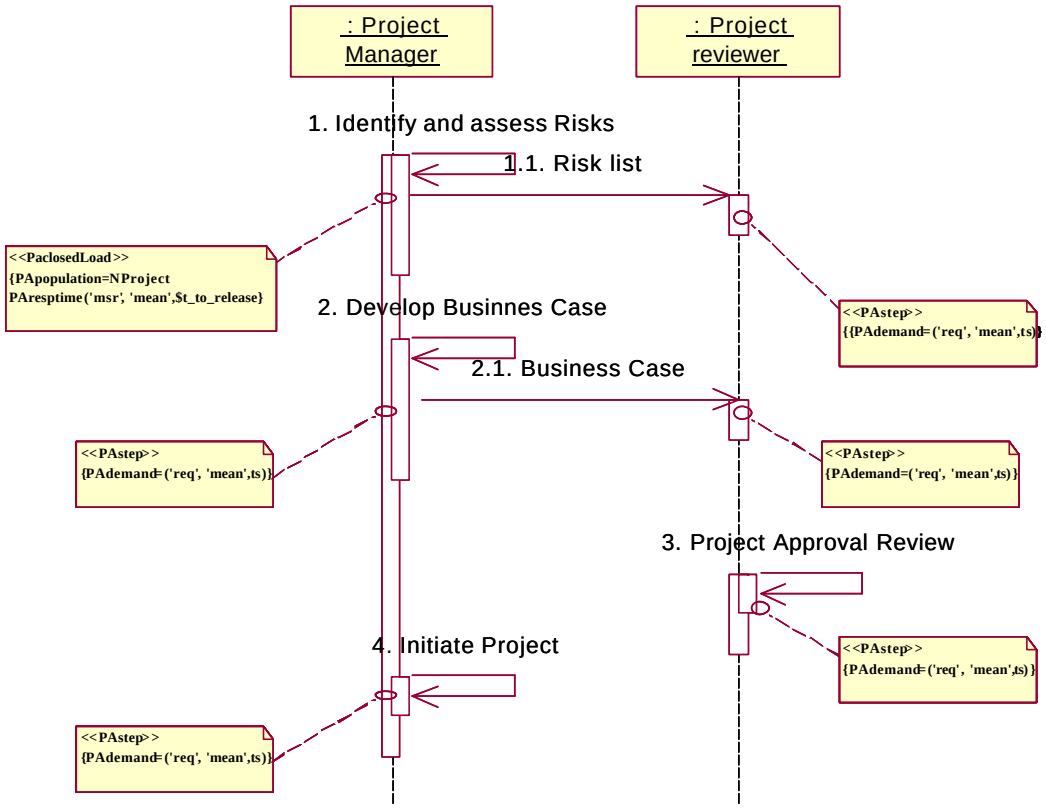


Figure 9 Activity "Conceive a new project" of the workflow Project Management.

The Propean modelling of RUP proceeds with the identification of the organizational structure of the system, i.e., architecture definition. As for system functionality, the architecture definition follows an iterative process describing the decomposition of the system into parts that interact through interfaces, relationships, and constraints. First of all we therefore describe as subsystems the Roles Set called Analysts, Managers, Developers, Testers and Additional Roles and we define the interfaces they use. In this case, the attributes of the interfaces are the set of exchanged documents.

The subsystem definition is then refined associating a class to every role and describing the interfaces they use. For every class the attributes represent the artifacts, and the methods are the activities in which the role is involved.

The UML description of RUP derived so far represents only the static structure of the process and is the common starting point for applying Propean to different real situations. The Propean user, typically a manager, starting from this process framework, must adjust and characterize it with respect to the specific needs,

peculiarities and constraints of his/her organization. In particular, as described in the next section, he/she has to identify the dynamic structure of the process and express the sequential flow of activities in the different RUP phases.

4.6.1.2 RUP Customization

In the previous section we briefly explained the incremental process adopted for deriving the UML model of the static structure of RUP. Here we discuss the required modifications to the steps of the Propean procedure presented in Section 4.4 in order to customize the RUP process to the specific organization needs, and to derive successively a queueing network based model for making predictions.

1. Manager: Analysis

For every phase of the RUP process (Inception, Elaboration Construction, and Transition) the manager, with reference to the UML static model described in the previous section, must specify the flow of the activities involved using the Activity Diagrams (ADs). In each of these diagrams, the decisions and conditions as well as the parallel execution of the activities must be shown. In particular the manager should decide: to possibly suppress some of the RUP activities of entire workflows according to the specific development needs of his/her organization, or to specify how many iteractions must be performed for each phase. In Figure 10 we report an example of a developed AD. In this case the activities of the different workflow, each one associated to a representative SD, are considered to be a sort of "building bricks" that the manager fits into the activity diagrams for describing the overall structure of the development process.

2. Manager: Modelling

As mentioned in Section 4.4, the manager must describe the organization structure in a Deployment Diagram, DD in which the nodes refer to both classical resources (device, processor, database) or personnel team. Figure 11 shows an example of an annotated DD.

In addition the manager must also specify the associations between roles and personnel. The RUP modelling supplies the manager with a Class Diagram with roles specialization; therefore he/she has only to reorganize the association between the different classes according to the organization exigencies. For example in Figure 12 is the class "designer" is associated which a real person (therefore it becomes a superclass) who can also assume the roles of Design reviewer, Database designer and so on (the subclasses)

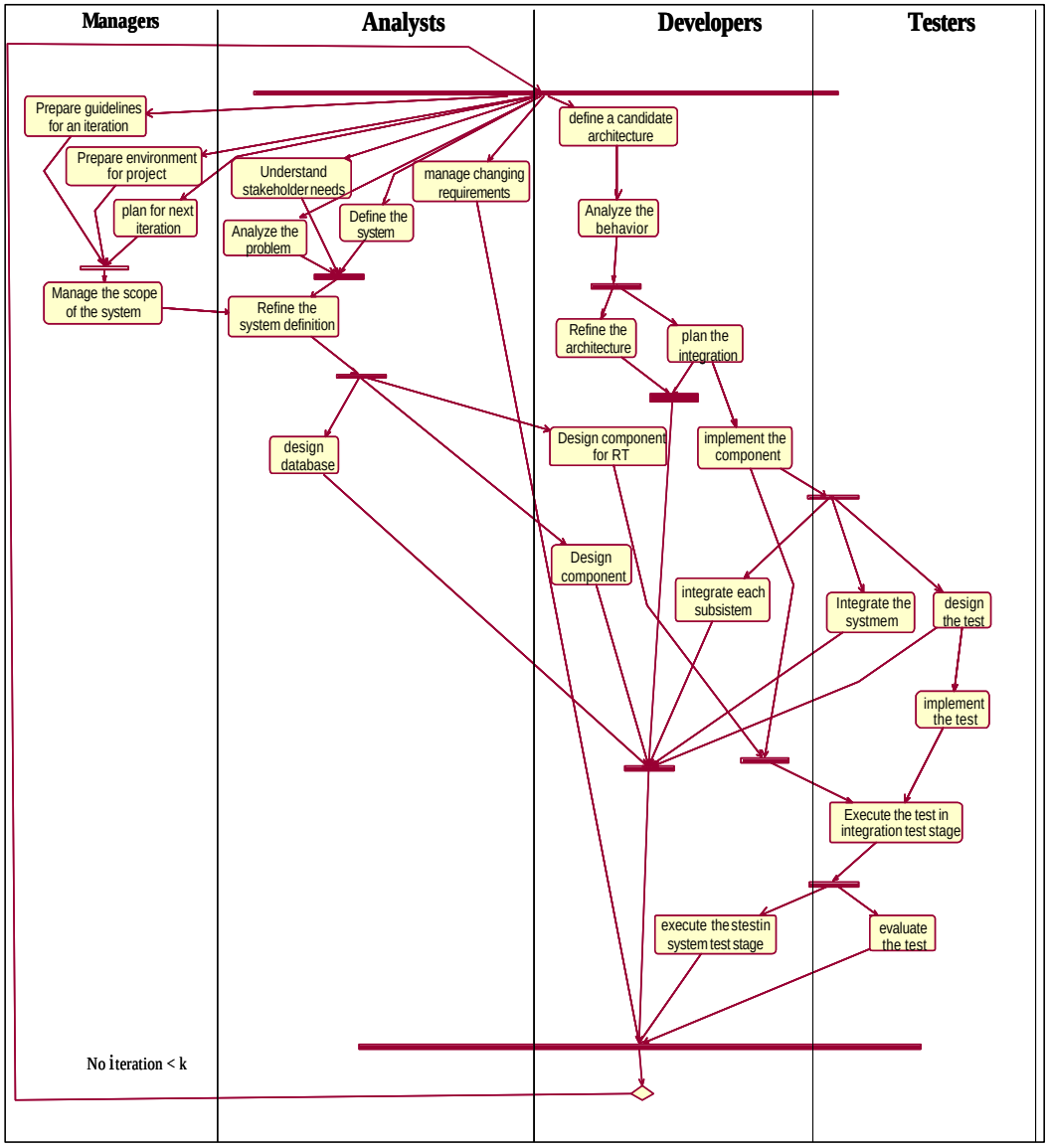


Figure 10 Activity Diagram relatives to the Elaboration Phase

3. Manager: Model annotation

In this step the manager must better specify the activities belonging to each Activity Diagrams developed in Step 1. Every activity is in fact associated with an annotated SD; therefore, the manager has only to refine the parameters or values of the stereotypes of the SDs description as mentioned in Section 4.4. The Figure 9 is an example of an annotated SD.

4. Automatic: SPE models generation

The SPE model is derived as explained in Section 4.4.1 considering the Activity Diagrams as well. It includes a model for the planned activities obtained by each

involved SD and Activity Diagram (the SM based on EG), and a model for the involved teams (the MM based on EQNM).

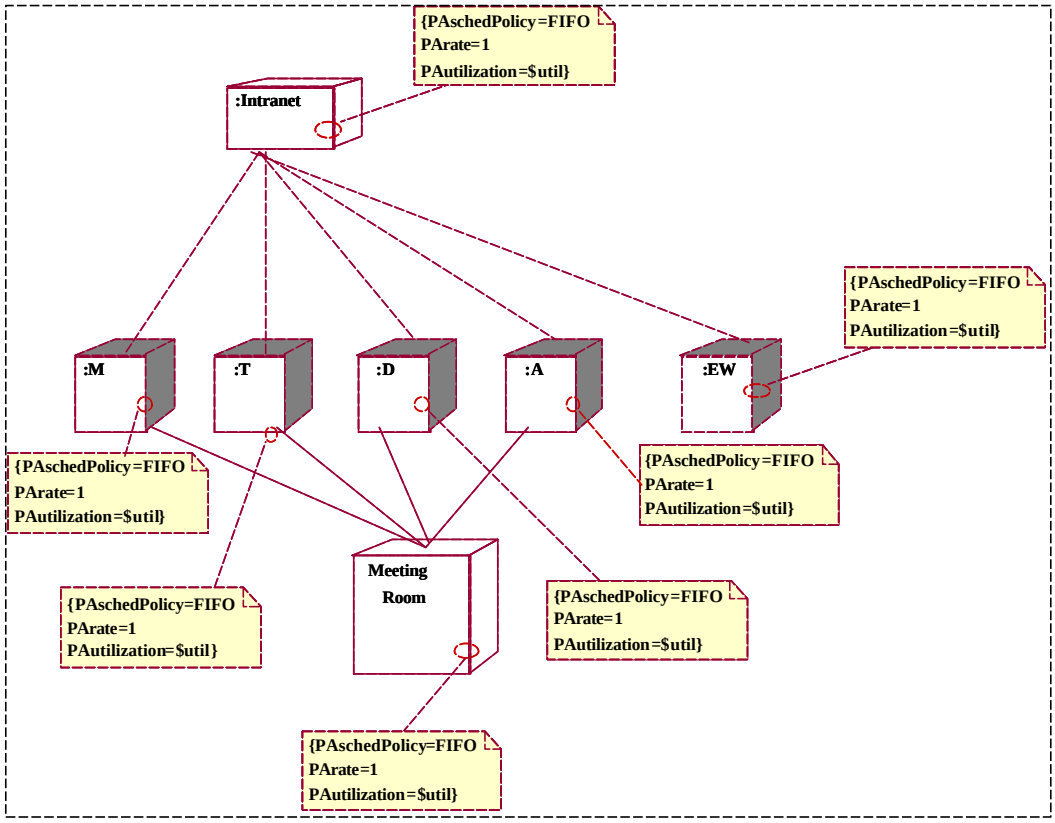


Figure 11 Deployment Diagram

5. *Automatic: Model evaluation*

The same of Section 4.4: the EQNM obtained in the previous step can be solved to obtain the desired results

6. *Manager: Analysis of results*

The same of Section 4.4: the results obtained in Step 5 are analysed by the manager, who can decide to go back to the previous steps.

4.6.2 An Example of Propean Application

To see how Propean works, we will apply it here to a case study consisting of a hypothetical project development. Although this example is built ad hoc for illustrative purposes, its organization and the assigned parameters (people involved and planned time for the composing steps) faithfully reflect the management

practices of a real-world organization. We describe the project here and in the next section provide the results obtained by applying Propean.

The system to be developed is composed of two large subsystems, A and B. Subsystem A consists of three components, and subsystem B of two components. For clarity, the development process of this system is represented in Figure 13 by means of an Activity Diagram.

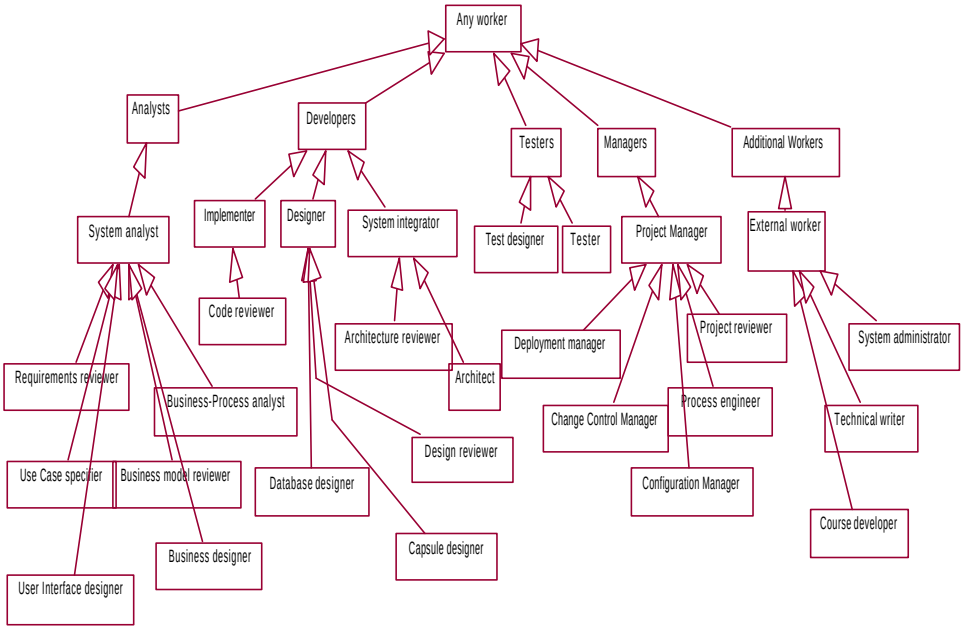


Figure 12 Class diagram relative to the Roles distribution

To apply Propean, we have modelled the process parts relative to every node of this AD using the "bricks" methodology overviewed in the previous section. Each node is detailed into the RUP activities referring to it and modelled by means of one or more SDs. The various activities corresponding to the nodes of this AD are distributed among the four phases of RUP, and associated with the activities forming the various workflows. For instance, considering the AD of Figure 10, the activity "Define the system" in the Analysts swimlane implements the activity "System Design" of the AD of Figure 13. In other words, we tailored the generic RUP model in Propean to the specific needs of this project, which is a small one, and therefore we obtained a rather simplified RUP configuration. Then, according to Step 3 we annotated the SDs with the estimated duration of each activity. The estimations were

made by an industrial manager as an average “guess” based on ERI standard parameters, assuming: the size (in terms of code lines) of the components as small, a medium system complexity, the existence of a design basis, a well-known technology, and medium competence of project team. Moreover, we assigned the personnel who will have to carry on the planned activities.

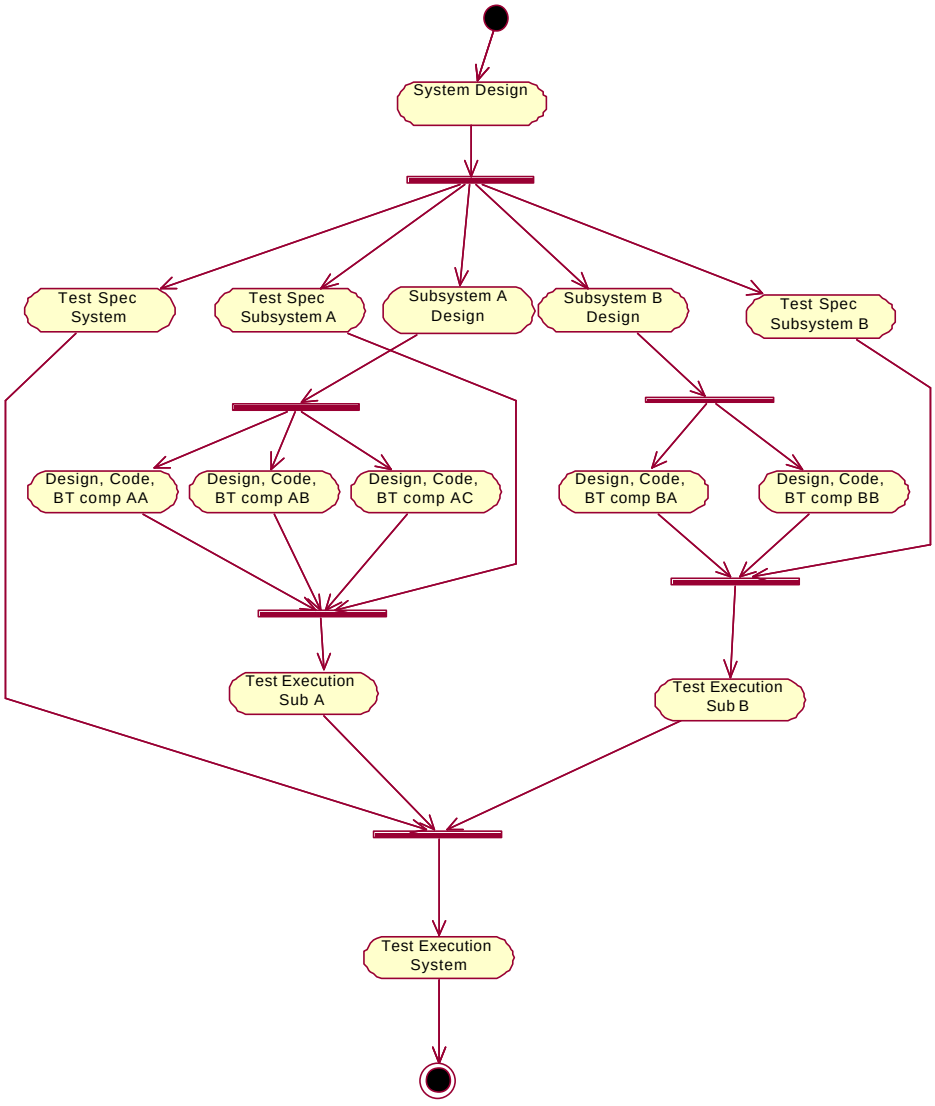


Figure 13 Development process representation

Specifically, we assumed as the initial configuration for analysis the following, referred to as Conf_I:

- 1 Project Manager (PM)
- 1 System Analyst (SA)

- 1 Designer (D)
- 1 Implementer (I)
- 1 System Integrator (SI)
- 1 Test Designer (TD)
- 1 Tester (T)
- 1 External Worker (EW)

Note that this is the initial configuration; during the analysis with Propean it may be the case that the hypothesized configuration shows itself as inappropriate and alternative configurations are found to be more effective. One of the objectives of Propean is to assess whether the personnel utilization is adequate with respect to project needs.

4.6.3 Analysis of Results

Here we show and comment on some of the estimations that can be automatically obtained by means of Propean.

At a first run, we derived the expected time to completion for the initial configuration Conf_I shown above. The estimated times for the four RUP phases considering the development of the system stand alone, i.e., without any concurrent project development which could compete with this for resources, are given in the first column of Table 9. Conf_I results are given in the white part of the table.

But the real potential of Propean is that we can also make estimations in the more realistic hypothesis that people are not dedicated full-time to this project, and that other RUP processes are going on concurrently.

For instance, in the table, we also show the estimated time to completion of each process, assuming that two or three processes are concurrently running; we can also make hypotheses on how the processes are interleaved. So for instance we considered the case that the 2 or 3 processes started contemporaneously (0 days of displacement), or that a process starts 30 days after the preceding one started (30 days of displacement). The times are given in days for one process.

As is plausible, we find that the concurrency among more projects brings large delay on the schedule: a project alone takes 188 days to complete, but if there are two projects to manage contemporaneously, it will take almost 90 days more; this delay is a little diminished if some displacement is inserted between them, so that people can finish one task on a project before being engaged in the other one; but as the Elaboration phase in this project is very long, then we see that the advantage is not

big and involves the first period of the project. Things get worse if we add more projects in parallel. The other interesting feature of Propean is that we can look at how the involved teams are charged, and identify the bottlenecks in people configuration.

		Number of Concurrent Projects				
	Phases	1	2 (0 days displ.)	2 (30 days displ.)	3 (0 days displ.	3 (30 days displ.
Conf_I (8 pp)	Inception	28,81	44,72	40,25	57,63	54,33
1 PM, 1 SA	Elaboration	104,58	155,76	141,35	194,21	187,23
1 D, 1 I, 1 SI	Construction	29,56	35,54	35,79	46	44,32
1 TD, 1 T	Transition	25,33	38,30	36,68	49,5	48,61
1 EW	Total	188,54	275,14	254,62	349,67	337,25
Conf_A (8 pp)	Inception	28,89	43,42	40,18	56,58	50,38
1 PM, 1 SA	Elaboration	91,88	127,06	120,24	156,63	142,82
2 D, 1 I, 0 SI	Construction	25,73	33,36	33,26	42,04	40,97
2 TD, 0 T	Transition	25,88	37,96	36,53	50,16	47,08
1 EW	Total	172,53	241,49	230,62	308,02	283,07
Conf_B (6 pp)	Inception	28,81	43,1	40,08	56,94	50,84
1 PM, 1 SA	Elaboration	104,58	152,57	142,96	190,43	179,26
1 D, 1 I, 0 SI	Construction	29,56	37,72	36,03	47,85	45,46
1 TD, 0 T	Transition	26,18	18,14	36,26	49,5	47,79
1 EW	Total	189,38	271,41	255,88	349,47	325,02

Table 9 Propean estimated times (in days) for the RUP phases

In Table 10, we report the utilization rate of people (in the interval 0-1, where 0 means idle and 1 means the person time is saturated at its maximum). We can see that in the planned workflows, the System Integrator SI and the Tester T are idle most of the time, as their utilization rate is very low compared to the other people involved. Therefore, we analyse how the RUP performance would change if we decide to assign different activities to these two people: in the light grey part of the two tables we report the results obtained assuming a different configuration, called Conf_A, in which the person who acted as a System Integrator before is now assigned part of the activities of the Designer, while the person acting in Conf_I as a Tester is given a Test Designer role here. As is obvious, we are also trying to consider the expertise of people, and we are reconfiguring them accordingly. The reconfiguration allows the manager to save 16 days in the schedule for the stand-alone project, and even more days in the multiproject scenarios. Moreover, if we now look at the utilization rate of people, we see in Table 10 that the effort is more evenly distributed among people.

Number of Concurrent Projects					
Personnel	1	2 (0 days displ.)	2 (30 days displ.)	3 (0 days displ.	3 (30 days displ.)
PM	0,09	0,13	0,13	0,15	0,15
SA	0,33	0,45	0,44	0,55	0,52
D	0,44	0,6	0,59	0,71	0,68
I	0,31	0,44	0,42	0,55	0,51
SI	0,005	0,006	0,007	0,009	0,008
TD	0,25	0,34	0,33	0,4	0,38
T	0,06	0,09	0,09	0,11	0,1
EW	0,13	0,17	0,16	0,2	0,19
PM	0,10	0,15	0,14	0,18	0,17
SA	0,36	0,52	0,47	0,62	0,59
D	0,24	0,35	0,33	0,4	0,4
D	0,24	0,34	0,32	0,43	0,39
I	0,340	0,49	0,47	0,58	0,57
TD	0,16	0,24	0,22	0,28	0,28
TD	0,16	0,24	0,23	0,29	0,27
EW	0,13	0,19	0,18	0,23	0,22
PM	0,10	0,14	0,13	0,16	0,15
SA	0,33	0,46	0,43	0,54	0,52
D	0,44	0,62	0,59	0,71	0,69
I	0,310	0,44	0,42	0,53	0,51
TD	0,32	0,44	0,42	0,51	0,5
EW	0,12	0,17	0,16	0,2	0,21

Table 10 Propean estimated utilization rate of people for the RUP phases

We have also investigated a different hypothesis, called Conf_B: given that SI and T are not doing much, we move them away from this project, redistributing their tasks to the other people (SI task to D, and T task to TD) i.e., we only assign 6 persons to the process. In this case the results are shown in the dark grey part of the two tables. We still get a more even distribution of effort among people, while the time to completion remains quite similar to that of Conf_I (i.e. we get almost the same time with less cost).

Thus in conclusion the results of Propean analysis for this case study leave the project manager with either of the alternatives: getting the project completed in a shorter time and more rational employment of resources with 8 people, or getting it completed in almost the same time of the initial configuration, but employing only 6 people. Clearly that the initial configuration is not an effective choice. In addition, Propean provides quite reliable estimates of schedules, based on the manager estimations for the single activities.

Summary

In this section we present an innovative approach called Propean, useful for defining a suitable test plan in a multiproject environment in which resources and personnel are shared among products. Propean constitutes an integrated approach in which managers, by using familiar notations and tools, can both define UML models

of the flow of activities to be performed during the development and of the tasks to distribute among personnel, and automatically derive the measure of interest. Propean translates the developed models in a format that is processable by standard performance analysis algorithms, so that a solver of these last can be applied to obtain the desired results.

Therefore we presented the procedural steps required for applying the Propean methodology and showed how the well-known techniques from performance analysis can be usefully and quite naturally adapted to tasks of relevance for software managers: assessing the time to completion of specified activities, handling personnel multitasking during different projects, optimising the workloads in development cycles, deciding about product release, and similar issues.

We demonstrate also how the use of Propean methodology can be extended to management of the other development phases as well as for the organization of the entire development process.

PART 3:
STRATEGIES FOR TEST CASE GENERATION

5 An Automated Approach to UML-Based Testing

Preface

The testing phase is an expensive but essential part of development, which must be well-organized and defined. Unfortunately often due to time or cost constraints it is not developed properly or is even skipped. Therefore methods and tools that facilitate and reduce the effort (time and/or cost) due to the management and the control of this critical phase are necessary. In the previous Chapter we presented the Propean approach, which automates the definition of a Test Development Plan before the effective launch of the testing phase. This document establishes the time-scheduling of the different activities and the resources and personnel assignment to the different tasks of the testing phase.

Then when the testing phase effectively begins, based on of the financial plan established, the test cases must be defined and distributed among the functionalities of the system to be tested, and subsequently executed. This is another critical point for the testing phase management. Generally, it is not easy to decide both the functionalities on which the testing effort should be concentrated and the amount of test cases to dedicate to each of them. Wrong decisions could increase the overall cost of the testing phase and the time required for its completion considerably.

We propose for this reason, in this Chapter an integrated, practical and automatic approach called Cow_Suite, which is prototyped in a tool, for generating and planning a suitable set of test cases, starting from the UML documentation. Cow_Suite includes both a method for deriving the test cases from the UML system specification and a strategy for distributing test cases among system functionalities (test cases prioritisation and selection).

One of the peculiarities of Cow_Suite is that it can be applied to real-world projects not only during the testing phase but, more importantly, from the early stages of system analysis and modelling, as soon as some UML diagrams have been

defined. Managers can therefore derive estimation, in terms of the number of test cases to be executed, of the effort required for completing the testing phase very early in the development process. Cow_Suite can thus be used in combination with Propean to derive the estimation values required for the application of this approach. However in this Chapter we concentrate mainly on the use of Cow_Suite alone, only suggesting in certain sections when integration with Propean could be possible.

In this Chapter we present the outline of the Cow_Suite approach (Sections 5.4, 5.5) and a brief description of the prototyped tool in its current status (Section 5.6). In particular we describe the Cow_Suite application in two case studies, one taken from the literature (Section 5.7) and the other provided by a real software developer (Section 5.8).

5.1 Cow_Suite Point of View

In this Chapter we concentrate on the methodologies for conformance testing from UML models. Until very recently UML-based testing has not received the attention it deserves and few methods have been proposed so far. In particular many of them demand too much on the developer's side and cannot be easily adapted to a real industrial context. They require either a rigid and meticulous modelling process, or often address a low-level test stage, or cannot scale-up to deal with large system portions. The reason is that UML was not created with testing purposes in mind, and does not readily allow for the rigorous analysis that is needed for automatic test derivation. This issue is referred to as the UML testability question in [BL01]. For test engineering a trade-off must be found between test thoroughness and cost.

Our response to this problem is an integrated, practical and automatic approach to the generation and planning of UML-based test suites, which can be applied to real-world-sized projects since the early stages of system analysis and modelling [BBM02]. To this end, we have developed a methodology and a prototype tool, called *Cow_Suite*, for COWtest pluS UIT Environment. As the name implies, the methodology implemented by Cow_Suite combines two original components: a method for deriving the test cases, called *UIT* (Use Interaction Test) [BB00], and a strategy for test prioritisation and selection, called *Cowtest* (Cost Weighted Test Strategy) [BBM01]. These two components work in agreement, as Cowtest helps decide which and how many test cases should be planned from the universe of test cases that UIT could derive for the system under consideration. UIT automatically generates test suites for the high-level test stages, encompassing system and

integration testing at various levels. Each generated test suite focuses on a functional portion of the system, as interactively selected by the tester on a structure of the suitable UML diagrams.

Thus Cow_Suite takes different position from other approaches, and proposes a pragmatic way of conceiving UML-based testing. This position of Cow_Suite can be summarised by its features of usability, timeliness, incrementality and scale, which we have pursued in an organic manner since the very inception of the approach. We explain one by one the meaning of these four features:

- **Usability:** where other methods require to augment the UML specifications with specific annotations to facilitate the test derivation, or to translate the UML diagrams into an intermediate notation that the methods can process (see next section), the leading principle of the Cow_Suite approach is to use exactly the same UML diagrams developed for analysis and design for test planning, without requiring any additional formalism or ad-hoc effort specifically for testing purposes. For us usability means that it is the test methodology that, as far as possible, adapts itself to the modelling notations and procedures in use, and not vice versa;
- **Timeliness**: according to the good software engineering principle that test planning should start as early as possible in the development cycle, a restricted set of minimal preconditions is assumed in order to start applying Cow_Suite (see Section 5.3.1). Typically, in the early design phases not all relevant scenarios are yet specified and the UML diagrams are defined at a high abstraction level, with several of them sketchy yet. While other methods require a complete and quite detailed set of UML diagrams, Cow_Suite can already begin outlining a test plan even at these early stages. Of course, the plan will be as abstract as the processed diagrams and is progressively refined as the diagrams are enriched with more information (see also the incrementality feature below);
- **Incrementality**: Cow_Suite was conceived for system and integration testing, (Chapter 2), which are typically conducted in an incremental fashion, considering progressively larger parts of the system and addressing, at each incremental step, the functionalities and the interactions that are relevant at the level considered. In Cow_Suite, the tester interacts with the tool in order to determine the integration stage for which the test suite should be derived (or, which elements of the UML model should be tested). Then, taking as a reference the corresponding UML diagrams, the UIT method derives the test cases at a specification granularity

corresponding to the degree of detail at which the considered diagrams are modelled. We are not aware of other UML-based test methods explicitly addressing incremental testing

- **Scale**: Cow_Suite trades thoroughness for comprehensiveness: as we intend to address UML-based testing of real-world systems in a practical, efficient way, we provided the capability to manage big test suites, keeping their sizes and functional coverage under control, via the Cowtest component. Other authors have proposed thorough and meticulous algorithms for deriving detailed test cases (see next the section), but often these methods either cannot scale up to handle the many big UML diagrams that are needed to model huge, complex systems, or would result in an unfeasibly large set of test cases. In contrast, the combined usage of Cowtest and UIT permits to derive a feasible number of test cases while keeping the coverage of functional areas as wide as possible

5.2 UML Testing: an Overview of the Literature

Even though UML is widely employed in industry and research, very little of the literature so far has addressed its use in the testing phases. In [WI99] the author brought up several possible issues that should be solved for effectively applying UML for testing purpose such as lack of detail and features of the UML models developed during the design and implementation phase. In particular in [EW03], the authors focus attention on the improvements of the UML requirement models to be used for testing purposes. We provide here an overview of the literature, presenting the solutions derived both from the academic and commercial environment (respectively Section 5.2.1 and 5.2.2), with the aim of placing the Cow_Suite tool in this context.

5.2.1 Academic Response

In this section we briefly present the approaches and tools available in the research area for the UML testing. In particular we differentiate them into two groups: those which require translation of the UML diagrams into an intermediate formal description (Group A) and those which requires annotation of the UML diagram with further (formal) information (GROUP B).

As stated in the previous section Cow_Suite differentiates both types of approach, because it does not require any additional formalism or ad-hoc effort specifically for testing purposes.

5.2.1.1 GROUP A

- **UMLAUT** (Unified Modelling Language All pUrposes Transformer) [JGP98] [UMLA]: it derives tests by translating the UML diagrams into an intermediate formal description, which can be processed by tools already constructed for different methodologies and adapted to the UML specifications. In particular UMLAUT transforms the UML representation of the system into a form suitable for validation within their VALOODS framework (VALidation of Object Oriented Distributed Software), which comprises a validation engine that will exercise the actual validation.
- **Offutt and Abdurazik approach** [OA99]: in this specific case the UML State Diagrams are translated into formal SRC specifications, from which input data for unit testing are automatically generated The same authors have presented in [OA00] a model for performing static analysis and generating tests inputs from a formal design description of collaboration diagrams specifications. The paper novelty was that tests could be generated automatically from the software design, which is also the leading criterion of the Cow_Suite tool, rather than the code or the specifications. Moreover the authors defined both static and dynamic testing criteria of specification-level and instance-level collaboration diagrams. These criteria allowed formal integration tests to be based on high-level design notations.
- **Liuying and Zhichang** approach [LZ99]: the authors propose deriving test cases from UML Statecharts, exploiting a formal semantic constructed for UML Statecharts. The method presented can automatically generate and select test cases from UML Statecharts in the context of object orientation, which will detect errors early in order to improve software quality. It is based on the Wp-method [FBK91], which deals with hierarchy and concurrency in structural context.
- **Kim et al. approach** [KHC99]: the author discusses the application of UML state diagram for class testing. To this purpose a set of coverage criteria is proposed based on control and data flow in UML state diagrams and the generation of test cases satisfying these criteria from UML state diagrams is shown. In particular they propose a transformation method from state diagram into extended finite state machine and flow graph. The transformation consists of flattering the hierarchical and concurrent structure of states and eliminating

broadcast communications, while preserving both control and data flow in the UML state diagram

- **Tsai et al. approach** [TSP99]. This approach is focused on class testing. The method utilises state machines in order to produce threaded multi-way trees, which are referred to as inspection trees. Inspection trees can be used to generate test cases and parse test results files. This allows the authors to determine whether the classes under test contain errors. The algorithms for the creation of inspection trees and the examination of the test results file using an inspection tree are described.
- **SCENT** (SCENario-based validation and Test of software) [RG00]. This is a method supporting requirements elicitation, analysis and definition by creating scenarios in a structured way, validating the scenarios with the customer/user and formalizing them into statecharts. From the statecharts, test cases, specifically designed for integration and system testing, are derived in a systematic manner by covering every transition.
- **Mayrhauser et al. approach** [MFS00]: The authors describe an approach to black-box test-generation in which an AI (artificial intelligence) planner is used to generate test cases from UML Class Diagrams. In particular these diagrams are used to derive test objectives and a domain theory which are then transformed to planner representations and given as input to the planner. The planner uses the problem description to generate a test suite that satisfies the UML-derived test objectives.
- **Graubmann and Rudolph** [GR00] the authors show that the UML-Sequence Diagrams can be seen as an object-oriented variant of the ITU-T standard language Message Sequence Chart (MSC) which is very popular mainly in the telecommunications area. Therefore, they include the MSC inline expressions and High Level MSC (HMSC) into Sequence Diagrams. In this approach the High Level MSC are used for formalizing and structuring the construction of scenarios for use cases in the form of HyperMSC, and then also employed as a basis for the specification of test cases.
- **Chevalley and Thévenod-Fosse approach** [CT01]: the authors proposed a probabilistic method, called statistical functional testing, for the generation of test cases from UML state diagrams, using transition coverage as testing criterion. In particular the emphasis is placed on defining an automatic way to produce both

the input values and the expected outputs. The technique is automated with the aid of the Rational Software Corporation's Rose RealTime tool [RSC].

- **Antoniol et al. approach** [ABP02]. This paper is focused on state-based class testing. The authors, considering the derivation of test sequences by covering all round-trip paths [BI99] in a finite state machine (FSMs), investigate this strategy when used in the context of UML statecharts. In particular, based on a set of mutation operators proposed for object-oriented code, the authors seed a significant number of faults in an implementation of a specific container class and investigate on its effectiveness in detecting faults
- **Harel and Marelly approach** [HM03]. The authors described a powerful methodology for scenario-based speci?cation of reactive systems. The approach is supported and illustrated by a tool (a play-engine). As the behaviour is played in, the play-engine automatically generates a formal version in an extended version of the language of live sequence charts (LSCs). As they are played out, it causes the application to react according to the speci?cation. In particular the Play-in is a user-friendly high-level way of specifying behaviour while the play-out is way of working with a fully operational system directly from its inter-object requirements. This approach can be applied to many stages of system development, including requirement engineering, speci?cation, testi ng, analysis and implementation.

5.2.1.2 GROUP B

- **Hartmann et al. approach** [HIM00]. This is a Siemens Corporate Research approach where the developers first define the dynamic behaviour of each system component using Statecharts; the interactions between components are then specified by annotating the Statecharts. Test cases are then derived from these annotated Statecharts using a test generation engine, and executed with the help of a test execution tool.
- **TOTEM** Testing Object-orienTed systems with the unified Modelling language) approach [BL01]. It supports the derivation of functional system test requirements, which are then used to then to derive test cases, test oracles and test drivers. The approach is mainly based on the use case sequences and uses the sequence or collaboration diagrams associated to each use case, transforming them into regular expressions, and the class diagram. In this process class

invariants and a detailed formal description of UML as well as a rigorous use of OCL notation are required.

- **SCENTOR** [WM01]. This is a research prototype aimed at supporting the generation of scenario-based testing using Junit as a basis. SCENTOR assumes a software engineering approach where lightweight UML modelling is part of the design process. Tests generated by SCENTOR are based on sequence diagrams contained within the UML model of the system. In particular this tool maintains the Extreme Programming focus on the production of source code. One of the purposes of SCENTOR, like Cow_Suite, is to reduce the required formal descriptions in the development process.
- **Latella and Massink approach** [LM01] the authors propose a formal testing framework for a behavioural subset of UML Statechart Diagrams (UMLSDs). In particular they define a new formal operational semantics and some proper testing pre-orders and equivalences which allow one to equate/distinguish systems on the basis of their interaction with the surrounding environment, abstracting from their internal structure. The purpose is to provide a way for effective automatic verification of testing equivalence of the statecharts, based on existing techniques and tools.
- **AGEDIS:** Automated Generation and Execution of Test Suites for DIstributed Component-based Software [AGE02] was developed in European project managed and coordinated by the IBM Israel Haifa Research Laboratory. Like Cow_Suite AGEDIS purposes are the automation of software testing, improving software quality, and reducing of the expense of the testing phase. In particular, for tests generation and execution it is firstly necessary to model the application under test in UML by the support of the tool Objecteering UML editor [OBJ] together with the AML (AGEDIS Modelling Language) profile for that tool. Then, annotate the model with further testing information, such as the coverage criteria, specific test purposes, and testing constraints.
- **Riebisch et al. approach** [RPG02]: The authors' scenarios and use cases, enriched by detailed behavioural information, for statistical test case generation. In particular they introduce an approach for generating system-level test cases based on use case models and refined by state diagrams. These models are then transformed into usage models to describe both system behaviour and usage. The method is supported by a XML-based tool for model transformation.

- **Pickin et al. approach** [PJT02] the authors investigate the use of formal validation in a UML-based development process. They present a method and a tool for automated synthesis of test cases from generic test scenarios and a design model of the application. The underlying "on the fly" test synthesis algorithms are based on the input/output labelled transition formalism.
- **SeDiTeC** [FL02]: this tool uses UML sequence diagrams, that are complemented by test case data sets consisting of parameters and return values for the method calls, as test specification. SeDiTeC supports specification of several test case data sets for each sequence diagram and automatically generates test stub for the classes and methods whose behaviour is specified in the sequence diagrams.

5.2.2 Industrial Response

We report here two commercial tools applicable to the UML documentation for different testing purposes. They range over many fields and are not specifically developed for integration testing.

- **The Rational tools [RSC].** The Rational Software Corporations provides various testing products which allow testers and developers to create robust software for a wide range of industries and platforms and enable them, by the use of automation and good practices. To create high quality software. One of them is Rational Suite® TestStudio® which automates functional, regression, unit, and performance testing and provides a seamless testing process, defect and change tracking, runtime analysis, software configuration management, requirements management and test management.
- **Analyst Pro** - Powerful UML Tool [ANA]. This is a tool for specification, analysis, design and testing of systems. Analyst Pro allows both specification of UML diagram as well as storing diagrams created with other tools. The test cases are generated from use cases and eventually update in case of changes in use case during the design.

5.3 Cow_Suite Methodology

The Cow_Suite approach consists of two components, working in a combined way: the Cowtest strategy, and the UIT method that will be described respectively in Sections 5.4 and 5.5 Here we present the minimal necessary requirements for Cow_Suite application and general schema of Cow_Suite utilization (Sec. 5.3.2), which shows how the two components, Cowtest and UIT have been integrated.

5.3.1 Prerequisites for Applying Cow_Suite

The leading criterion of Cow_Suite is the use of the same UML diagrams developed during specification and design, without imposing on the UML designers any additional formalism, or ad-hoc effort. The approach can be used in all the phases of the software development process, even though some diagrams have yet to be completed or refined. Of course, like any other test strategy, Cow_Suite needs to refer to a documented and systematic design process and for this reason we set some minimal requirements. However, they are very basic requirements, in no way test-specific: they establish a minimum discipline in design documentation that should be enforced in any standard software engineering process, and not only for the sake of testers.

Depending on their granularity, first we depict the more general prerequisites and then those strictly related to the development on UML documentation. In particular the latter are mainly inspired by the best practices and guidelines of the RUP process development [RUP, KR00].

General Requirements:

1. Cow_Suite is mainly based on the analysis of the Use Cases (UCs) and Sequence Diagrams[1] (SDs). In particular for organizing the UML element in a sort of hierarchy it is necessary to explicitly define associations and relations among the developed UCs, and between Actors and UCs, such as, for example, “uses” or “generalization” relationships.
2. It is important to keep track of how a UC is refined in the low-level design; this means specifying how a high-level UC, i.e., system functionality, is realized within the packages of the design model.
3. As the UIT method is based on an analysis of the SDs, the description of relevant scenarios are of course essential.

However, in early design phases, it is plausible that the UCs are defined at a high level, and many of them have to be completed; similarly not all relevant scenarios are elicited or documented. The Cow_Suite approach can be also useful under these conditions, because it can highlight points of weakness in the reference documentation. Specifically, it provides a picture of the project level of completeness and prompts the user for the revision or the completion of the unfinished diagrams.

[1] Collaboration Diagrams are also usable because, for our purposes, they contain the same information of Sequence Diagrams. Nevertheless, we only refer here to SDs analysis.

Detailed Requirements:

1. The design elements, such as Actors, UCs, SDs, CDs, defined during the requirement phases, and represented in the Use Case View, should not be designed alone, but in relation to other elements of the same view, for example by means of associations or relations.
2. Every UC defined in the Use Case View (high level UC) (Chapter 3), must have a corresponding UC in the Logical View (Chapter 3), stereotyped as <<use case realization>> with a dependency (stereotyped «realize») on the Use Case. This represents the refinement of the high-level UC in the lower-level design diagrams. SDs or CDs should be used for this purpose.
3. Every UC stereotyped as <<use case realization>> must be collected, in the Logical View, in a package called "Use Case Realizations". This in turn should contain a Class Diagram called "Traceability" in which the associations between the UCs and their realizations are defined. Referring to RUP the separation from the UC and its realization is necessary in order to allow the changes of the use case implementation without affecting the baseline use case.
4. Every Use Case Realization should contain references to the package/s that represent the implementation of the relative UC (referring to RUP the packages of the Analysis Model or the Design model). We suggest collecting them into a package called "Design Link" package, as will be explained in detail in Section 5.4.1.2.

5.3.2 Cow_Suite Usage

The application of Cow_Suite begins early in advance from the code definition from the analysis and design phases, and proceeds in parallel with them by analysing the UML project description for the derivation of the test cases. In this way, as the design evolves, a more refined and complete test plan is automatically derived. In this section we briefly present the steps leading from analysis of the UML description to test cases derivation. In particular we report in Figure 1 an Activity Diagram describing the actions required for the application of the Cow_Suite tool. This diagram is a sort of roadmap for following the application of the tool. The details of the various steps will be presented in the different sections.

In the figure the different colours of the activities show the separation between the test strategy (Cowtest) and the method used for deriving the test cases (UIT): respectively from Activity 0 to Activity 8 and from Activity 9 to Activity 12.

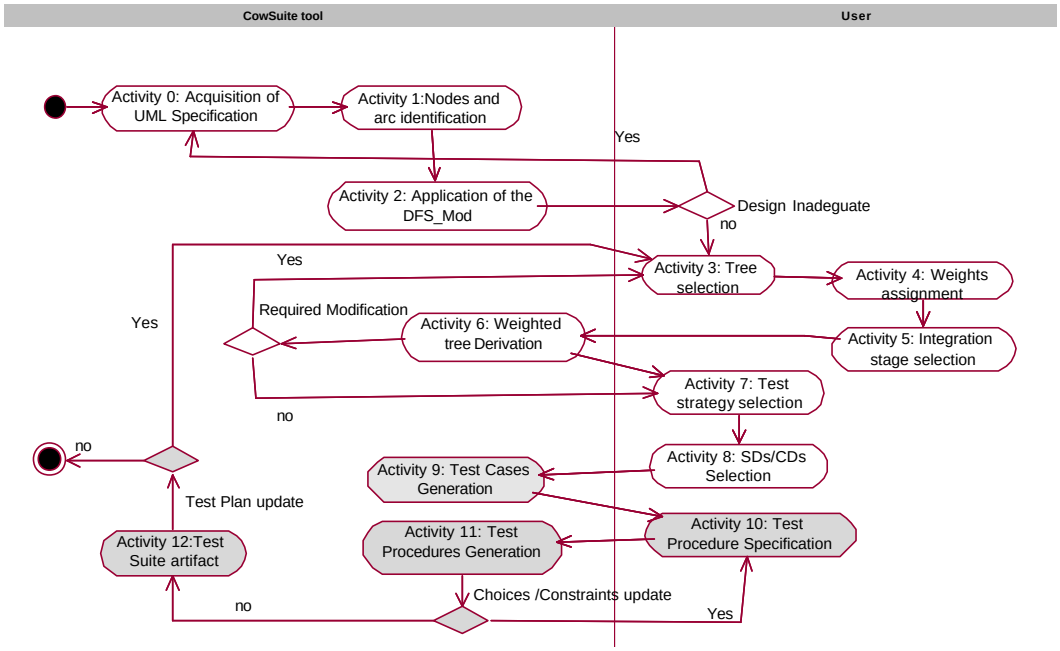


Figure 1 The Activity Diagram of the Cow_Suite usage

As shown in the diagram the activities from 0 to 2 focus on the acquisition of the UML project specification and on the organization of the design elements in a well-defined structure. This will be the basis on which the Cowtest strategy relies. The activities are completely automated by the Cow_Suite tool, which uses specific procedures and algorithms for the structure derivation (See Sec. 5.4.1.3).

From Activity 3 to Activity 9 user interaction is required both for assigning several specific parameters values and selecting the testing strategy to be applied. The Cowtest strategy can be applied into two different conditions: either the testing must respect a certain resource investment, which we translate in practice into fixing a number of test cases to be executed; or the testing has the purpose of covering a fixed percentage of system functionalities and therefore the test cases must be concentrated on them (See Sec. 5.4.3)

The UIT methodology is then applied for deriving the test case frameworks (Activity 8 and 9) (See Sec. 5.5). In particular, by using the information provided by the user the final test specifications, called Test procedures, are derived (Activities 10, 11 and 12).

As shown by the activity diagram, every step can be repeated several times during project development and consequently the final test procedures can be specified at different degrees of detail. Thus the proposed stepwise methodology can

be an effective support for test scheduling and planning which proceeds along with project development.

5.4 The Cowtest Strategy

In the following subsections we present a stepwise description Cowtest (Cost Weighted Test Strategy) which is a practical aid to managers for test planning. In particular, we distinguish two different test planning schemes: testing must respect a certain resource investment, which we translate in practice into fixing the number of test cases; or the test cases must cover a fixed percentage of functionalities. Accordingly, the Cowtest strategy can implement two approaches: a fixed number of tests or fixed functional coverage. The choice between either of the two is performed in Section 5.4.3 while in the next section the procedure adopted for deriving the basic structure on which the strategy relies is described.

5.4.1 Deriving The Basic Structure

A UML design consists of several diagrams containing various model elements, and forming the different views of the system. In this section we explain the procedure applied for organizing the model's elements into a defined structure. In particular the analysis starts from the diagrams collected in the Use Case View, considering in particular their mutual relationships, and proceeds with those of the Logical View. Referring to the activity diagram in Figure 1 in this section we describe the activities 0, 1 and 2.

5.4.1.1 Use Case View Analysis

As described in Section 3.1.2.1 the Use Case View is the main representation of the system, even if the coarser in terms of architectural description. Its purpose is the depiction of functionalities that the system must perform and the explanation of the interactions between the system and the external world. We begin the analysis of the Use Case View from the main Use Case Diagram, considering its design elements (Actors, UCs, SDs, CDs), and we proceed in an ordered way, with the other diagrams (if any) of this view. The purpose is to collect the design elements and the relations between them into two different sets, called "V" and "E", respectively. In particular every UC, Actor, SD or CD, is associated to a different node (also called "vertex") and is name inserted in the set "V". Contemporarily, every relationship, such as generalizes, extends, uses, etc., between the nodes is related to an oriented arc and

put in set “E”. The orientation of each arc follows the semantic relationship between the elements involved (not the graphic notation). For instance for the generalization relationship the arc orientation is from the parent to the child (as reported in the UML 1.4 semantics [UML]), i.e. in the opposite direction with respect to the associated arrow.

The nodes and the arcs are then organized into an oriented graph MG (V, E), called the *Main Graph*, representing a global description of the project. It may not always be possible to represent the design description with only a single graph. When some connections between the different model elements are missing, or there are some holes in the design, the vertices of the set V are split up into disjoint subsets and the Main Graph is disconnected into more subgraphs.

5.4.1.2 Logical View Analysis

As described in Section 3.1.2.2 the Logical View is mainly an architectural view of the system which constitutes a basis for its structure and organization. We use its information for three different purposes as described below: completing the Main Graph, defining a new graph called Design Graph, and introducing a specific package, called Design Link.

- **Completing the Main Graph**

Coming back to the test strategy, the information contained in the Logical View is used to upgrade and extend those collected during the analysis of the Use Case View. Therefore we first consider the Use Case Realizations package (Sec. 5.3.1), which may be created in either the Analysis Model or the Design Model or both. In particular we focus on the use case realizations belonging to its class diagram "Traceabilities" (Sec. 5.3.1), which are strictly related to the UCs of the Use Case View. Each of them in fact describes how the UC is realized within the Logical View in terms of collaborating objects. In particular, use case realization shows the implementation of the UC by creating a group of classes working together to describe the behaviour of the UC, and a set of SDs and CDs which explicitly show how the interaction among these classes evolves. The focus of the use case realization is therefore to separate the specifications of the system at requirements level, i.e. UCs of the Use Case View from the architectural design of the system.

In our test strategy we use these new design elements to augment the set V of the Main Graph definition. A different node is associated to each use case realization, or every SD/CD linked with it, and inserted in the set V. On the other hand each

relationship or <<realize>> dependence, between these design elements is related to an arc and inserted in the set E of the Main Graph definition. As for the relationships individuated during the Use Case View analysis, the arcs orientation follow the semantic relationship between the elements involved.

- **The Design Graph Definition**

The remaining design elements (packages and their components) of the Logical View are then analysed from a different perspective, for defining a new oriented graph called *Design Graph.* This is a graph DG(V', E') in which the set of vertices V' consists of all the developed packages or components and the set of arcs E' represents the dependences between these elements. The Design Graph is structured following a process similar to that used for the Main Graph. The construction starts from the Design Model package and proceeds in an ordered way, with the analysis of its packages, excluding the previously considered Use Case Realizations. In particular the design elements considered during this process are:

- The packages, each one associated with a different a node of set V'. If a package is further subdivided into sub-packages a node for each of them is also inserted in V'. Moreover, an oriented arc (from the high level package to the sub-package) is inserted in the set E' for each of the sub-packages.
- The SDs linked to the packages. A node in V' is inserted for each of them and an arc from the package to the SD is defined in E'.
- The package diagram, if any representing the dependences among packages. In this case a node of the set V' is associated to each of the involved packages (if not jet included in V') and an oriented arc, representing the dependences between the packages, is inserted in the set E'.

The Design Graph is therefore an oriented graph, showing the hierarchy and the dependences between the different packages or SDs. In other words it represents the organized structure of the components, sub-system, and other parts that will represent the architecture of the system to be developed. However, as already indicated for the Main Graph, it is possible that the Design Model analysis produces a disconnected Design Graph with sub-graphs or isolated nodes. Once again this is due to the incomplete or badly-defined UML project specification and it is not imputable to the process used for deriving the graph.

The Main Graph and the Design Graph differ as to the model elements they consider, but especially in the kind of information they collect. The Main Graph is, in fact, a high-level representation of the system: the UCs represent the functionalities

or the sub-functionalities of the system and the SDs the description of how the UCs are realized by the interaction between objects and actors. The Design Graph, instead, provides a lower-level description of the system, the packages represent the components or sub-components that will be implemented and the graph structure is a mapping of the project architecture.

- **Design Link Definition**

It would be extremely important to be able to connect the information contained in the Main Graph with those of the Design Graph. Considering a UC of the Use Case View representing a high-level requirement, it is generally extremely difficult to individuate the components, in the Logical View, that implement it. The problem is accentuated in the presence of dependences between packages or between the elements inside the packages themselves. In other words, considering the two graphs there is a lack of connection between the nodes of the Main Graph associated to the UCs and those of the Design Graph representing the packages or components, which implement them. It is worth noting that this deficiency is not due to the procedure adopted for deriving the two graphs but is imputable to the UML project specification. The compact description of the UML design provided by the Main and the Design graphs only further emphasizes this lack of information.

In the literature, this problem has already arisen and some solutions provided. In the ICONIX process [RS01], Rosenberg and Scott close the gap between the requirement level view and the detailed design view by introducing a new kind of diagram, called “Robustness Diagram”. This was originally introduced by Ivar Jacobson work’s [JCJ92] which included it in the UML standard as an appendage. It was depicted as an UML collaboration diagram, showing both the objects that participate in a scenario and how they interact with each other. In short, the Robustness Diagram as adopted in ICONIX is instead a class diagram, which shows object instances rather than classes and contains three kinds of objects:

- The boundary objects, which actors use in communicating with the system
- The entity objects, which are usually objects from the domain model
- The control objects which “connect” boundary objects and entity objects

We do not wish to provide a complete description of the Robustness Diagram here, preferring to refer the reader to the book of Rosenberg and Scott [RS01] for more details. However, inspired by the ICONIX process, we propose a solution compatible with the procedures adopted for the Main and Design Graphs derivation. Therefore we suggest including in each use case realization a specialized package

diagram, called the *Design Link,* collecting the list of the packages of the Logical View that implement the UC associated to the use case realization considered. The Design Link is therefore intended to fill the gap between the requirements and the detailed design. It represents a useful piece of documentation for both the people involved in the design and those who will implement the system.

When the Design Links are available, during the definition of the Main Graph, a node is associated to each of them and inserted in the set V; the same is done for the packages referred in the Design Link. Contemporarily the arcs, representing the connection between the use case representation and Design Link and the Design Link and these subpakages, are inserted in the set E.

Thus the Design Link and its packages represent the "glue" between the information of the Main Graph and the Design Graphs, which associate each requirement to the relative system components.

5.4.1.3 Trees Derivation

In the previous section we have described the necessary steps for deriving the Main and the Design Graph from the analysis of the UML documentation. Here we show the procedure adopted for deriving the basic structures of our strategy application (Activity 2 of Figure 1). It is worth noting that the analysis of the Design Graph is almost the same as those applied to the Main Graph; therefore we describe only the latter in detail. We want to isolate each interaction of the external environment with the system for the purpose of testing it separately. In the Main Graph this is translated into associating to each actor, responsible for one or more external stimuli, a different tree, which expresses the way in which the interactions are implemented in the system.

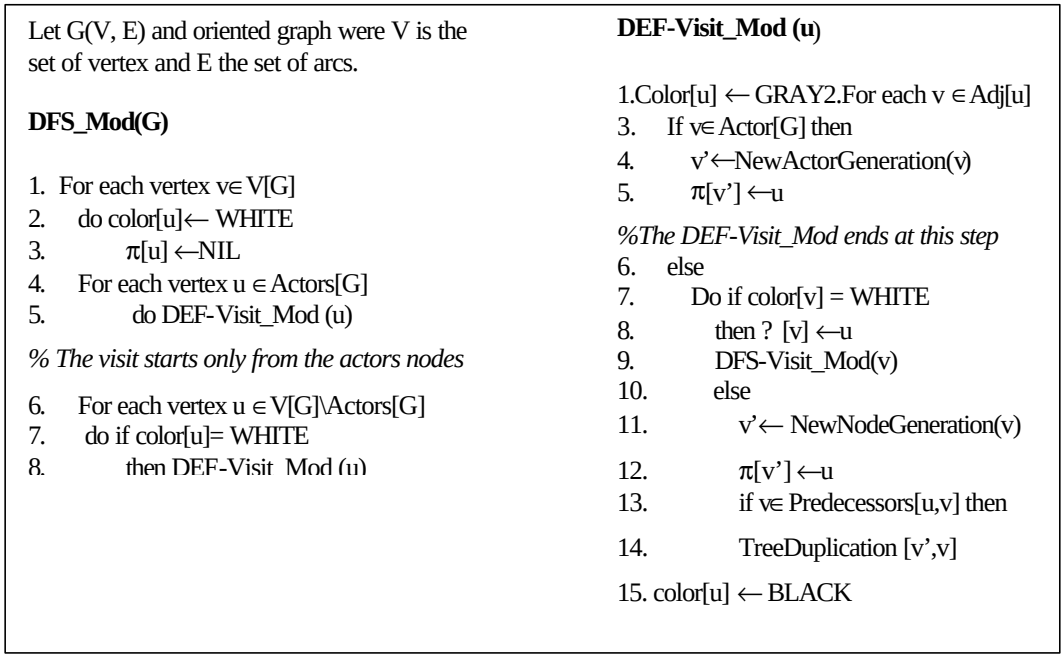
The trees derivation is performed by using a modified version of the Depth-First Search algorithm [CLR01] called DFS_Mod showed in Figure 2, which produces a forest of several Main Trees. In this section we describe only the main characteristic of the algorithm applied for the Main Trees derivation, since that for the Design Trees is almost the same. The trees obtained constitute a detailed documentation of what has been developed so far, highlighting the structural decomposition of the functions. Therefore considering the Main Graph G(V,E), obtained as described in the previous section, the DFS_Mod derives trees with these peculiarities:

- The root is always represented by an actor, who is a person (or external system) interacting directly with the system. The actor requests are therefore the functional stimuli for the system.
- The UCs at the first level represent the requirements, each associated with a different functionality that the system must realize. In particular a functionality could be in turn specialized or refined into sub-functionalities, which correspond to the UCs at the second level in the tree.
- The SDs /CDs (if any) at the second level of the tree describe the interactions and the exchanged messages among the objects belonging to one of the UCs at first level.
- Considering the *i-th* level of the tree, the UCs represent the description or the realization the sub-functionalities at upper level and the SDs/CDs the description of the objects' interaction of the UCs at *(i-1)-th* level.
- Some parts of the tree are opportunely replicated or marked: they belong to other trees or to repeated nodes and are signals of the presence in the Main Graph of cycles or of elements reused in more diagrams.

The trees derivation can also be applied in the anomalous situation in which the Main Graph is not connected. In this case the DFS_Mod algorithm produces a set of "anomalous trees" which can for instance either be represented as a single node (model element), or as a tree rooted in a UC instead of an actor. These trees, classified as "Not Linked", are not used in the strategy application. In Figure 6 we show a Main Tree, the Design Tree and the set of Not Linked elements derived by applying the Cow_Suite tool to a case study.

The advantages in reorganizing the design element in this hierarchical structure are numerous; we list some of them below:

- *Complete view:* Each of the derived trees (Main Trees, Design Trees) describes to the people involved in the development (specifically the project managers) the level reached in the functionality implementation. Considering for instance a Main Tree, the paths from a UC to the leaves represent the specification of the UC and describe its level of implementation in the design. The SDs encountered in each path depict, at different levels of detail, the behaviour of the functionality associated to the UC and specify the required interactions among the involved design objects.



Let G(V, E) and oriented graph were V is the set of vertex and E the set of arcs.

DFS_Mod(G)

1. For each vertex v∈ V[G]
2. do color[u]← WHITE
3. π[u] ←NIL
4. For each vertex u ∈ Actors[G]
5. do DEF-Visit_Mod (u)

% The visit starts only from the actors nodes

6. For each vertex u ∈ V[G]\Actors[G]
7. do if color[u]= WHITE
8. then DEF-Visit_Mod (u)

DEF-Visit_Mod (u)

1.Color[u] ← GRAY2.For each v ∈ Adj[u]
3. If v∈ Actor[G] then
4. v'←NewActorGeneration(v)
5. π[v'] ←u

%The DEF-Visit_Mod ends at this step

6. else
7. Do if color[v] = WHITE
8. then ? [v] ←u
9. DFS-Visit_Mod(v)
10. else
11. v'← NewNodeGeneration(v)
12. π[v'] ←u
13. if v∈ Predecessors[u,v] then
14. TreeDuplication [v',v]
15. color[u] ← BLACK

Figure 2 The DFS_Mod Algorithm

- *Incompleteness of the design:* The Not Linked elements evidence some incompleteness or weakness in the UML design. This could happen either if some parts of the UML design are not yet completely developed at the time of the tree derivation, or if some relation between the design elements has been forgotten or not specified during the analysis or design phase. In both cases, the set of Not Linked elements clearly reveals this situation to the people involved in development so that they can immediately take proper corrective actions.
- *Estimations for management:* One aim is to provide a strategy which can be applied at any time during the software development. Thus, if necessary, the people involved in development can automatically derive the different trees and therefore have a complete vision of what has been developed so far. This could be used for a rough estimate of the effort and time required for completing the project specification.
- *Organized documentation:* The derived trees represent an ordered and organized documentation continuously updated with the latest changes and complete in every part.

5.4.2 Defining a "Testing Profile"

So far the basic structure on which the Cowtest strategy relies, has been automatically defined by the Cow_Suite tool. Now the user must interact with the tool to specify the parameters or choices useful to adapt the test strategy and the test cases derivation to his/her needs. Referring to Figure 1 in this section we describe activities 3-6 which mainly consist of selecting one of the developed trees, assigning the weights to its nodes (Sec. 5.4.2.1) and choosing the proper the integration stage (Sec. 5.4.2.2). As in the previous section we discuss in detail only the steps for the Main Trees because those for the Design Trees are nearly the same.

5.4.2.1 Assign Weights to the Nodes.

Generally the various system functionalities do not have the same "importance" for overall system performance or dependability, and the testing effort should be planned and scheduled accordingly. Different criteria can be adopted in order to define what "importance" means for test purposes, e.g., component complexity, or usage frequencies (such as in reliability testing [MIO87]). Often, these criteria are not documented or even explicitly recognized, but their use is implicitly left to the sensibility and expertise of the managers. The basic Cowtest idea is that we ask managers to make explicit these criteria, and we provide them with a systematic strategy in order to use such information for test planning.

In particular, for each of the derived (Main) trees, managers are requested to annotate the nodes level by level with a value, belonging to the [0,1] interval, representing its relative "importance" with respect to the other nodes at the same level. This value, called the *weight*, must be assigned in such a manner that the sum of the weights associated to all the children of one node is equal to 1; the more critical a node the greater its weight.

Several criteria for assigning the importance factors could be adopted. Obviously this aspect in the proposed approach remains highly subjective, more in the realm of expert judgment than mechanisable methods, but here we are not going to provide a quick recipe on how numbers should be assigned. We only suggest expressing in quantitative terms the intuitions and information about the peculiarity and importance of the different part of the system to be developed, considering that such weights will correspondingly affect the testing stage. In our intuition, that only empirical evidence from the strategy usage history can legitimate, we believe that the strategy should be robust enough to moderate deviations.

However it is worth noting that the process of node annotation implies a beneficial side-effect: for assigning the appropriate values, the people involved in development are forced to reflect on the relative complexity of each functionality with respect to the context in which it is inserted. Consequently, they pay attention to the parts where problems could be more critical and become conscious of the importance of each node for the system development.

5.4.2.2 Integration Stage Selection and Weighted Trees Derivation

The Cowtest, and mainly the UIT method, are specifically developed for integration testing; thus before any test strategy, it is necessary to define the integration level at which the testing will be performed. Considering the previously trees derived, this means deciding which nodes to analyse for testing. Each level in a specific tree shows a different degree of detail of the system functionalities implementation and consequently a specific level of integration. Thus by excluding the root of the tree that for each Main tree is always an Actor node, we introduce the concept of an *integration stage*:

The **first integration stage** is represented by the UCs connected to the root node and the SDs/CDs (if any), which are the children of that UCs (hence they are at level 2 of the tree).

The ***i-th* integration stage** is represented by the UCs at the *i-th* level of the tree and every SDs/CDs, children of these nodes, situated at *i+1-th* level.

We decided to include the SDs at level *i+1* within the *i-th* integration stage, because they represent the interaction between the different components that realize the functionalities described in the UCs at i-th level of the tree.

Before applying any of the proposed test strategies, it is necessary to determine the integration stage at which the testing will be performed on the selected tree. Fixing the integration stage therefore means deciding which kind of information (nodes) to consider for deriving the test cases. Consequently the node selected will be: the nodes belonging to the fixed integration stage plus all nodes at higher levels, and in particular the leaves of the latter.

Having fixed an integration stage, it is now possible to use the weights assigned to derive for each node a relative importance factor, called *final weight*, in terms of how risky is that node and how much effort should be put into testing.

The *final weight* of every node is then computed as the product of the weights of all nodes on the complete path from the root to this node. It is the reference index for

choosing which tests to execute as will be described in the next section. Note that the sum of the final weights of the leaves is still equal to one.

In Figure 7 the numbers in square brackets are the final weights computed for the Main Tree of Figure 6.

5.4.3 Cowtesting

Following the steps described so far the trees' structure has been defined; for each of them a (different) integration stage has been selected and the final weight of each node calculated. Now it is necessary to determine test strategy to adopt for test case derivation. Referring to Figure 1 in this section we explain activityies7-8

We consider two different situations: either a certain number of tests is fixed, or the percentage of functional coverage is chosen as a stopping rule. The first is the case in which a certain test budget is available, which we translate in practical terms as a fixed number of test cases. In such a case, Cow_Suite allows us to derive the most suitable distribution of the available test cases among the functionalities developed. The second situation considered occurs when a certain percentage of functionalities must be covered for testing purposes. In this case using the tool, it is possible to determine by which functionalities are to be covered and the minimum number of test cases to execute. In both circumstances, an entire collection of test cases is automatically derived from each SD/CD by applying the UIT methodology as explained in Section 5.5. Here we present the strategy used for test selection and prioritisation in the two situations, respectively.

Similarly to the previous section we discuss only the procedural steps adopted for the Main Trees because the main difference with respect to the Design Trees is in the typology of the obtained test cases. Considering the Main trees, the test cases will be specifically designed for system or high-level subsystem integration testing (occasionally even for component testing). Those obtained by the Design Trees will be suitable for the low-level subsystem and component integration testing (rarely for unit testing). Therefore the test cases will reflect the degree of detail of the information collected in the different trees.

- **Cowtest_ing with fixed number of tests**

If a number NT of test cases is fixed (or, more plausibly, only a test budget up to NT tests can be afforded), our strategy can be used to select NT test cases out of the many test cases that could be conceived. In fact, using the final weight, called *nw*,

associated to each SD, the number *nt* of tests to be selected can be easily derived as: $nt = \lfloor nw * NT + 0.5 \rfloor$.

Clearly, a prediction of the number of test cases is just one half of the task. To make a more practical prediction in terms of man/hours, or required budget for testing, it would be necessary to estimate the cost of the various test cases, which is clearly not inconsiderable.

- **Cowtest_ing with fixed functional coverage**

Let us now consider the alternative case in which a certain percentage of functional test coverage (e.g. 80%) is established as an exit criterion for testing. In this case Cowtest can drive test case selection, by highlighting the most critical system functionalities and properly distributing the test cases.

For each SD representing a leaf at the chosen integration stage, its final weight, *nw*, is calculated as above. Then considering the fixed coverage C, the selection of the functional test cases to be run can be derived ordering in a decreasing manner the *nw**100 values and adding them together, starting from the heaviest ones, until C is reached.

Moreover using the final weights of the selected leaves, normalized so that their sum is still equal to 1, it is also possible to derive the minimum number of test cases required to reach the fixed coverage.

Considering that each test case required a certain amount of time, *t*, to be executed, this last feature was particularly useful in the early stages of the process development when Cow_Suite was applied in combination of the Propean approach, (Chapter 4) to estimate the overall duration of the testing phase.

5.5 Use Interaction Test

UIT, largely inspired by the Category Partition method briefly described in Section 5.5.1, was originally conceived for integration testing in order to systematically test the interactions among the objects, or object groups, involved in a SD/CD [BB00]. Within the Cow_Suite approach, we have integrated a modified version of the UIT method (for clarity referred to as UIT_sd), by which test derivation was carried out once for each SD/CD as a whole and not separately considering the objects involved. In this section, we only concentrate on the UIT_sd methodology, referring to Appendix B for mode details on the original UIT. However, before presenting it some definitions are necessary. Inspired by the RUP process [RUP] we distinguish between *Test Cases* and *Test Procedures.*

- A Test Case is the set of actions performed to test a possible objects interaction, with associated test inputs and execution conditions.
- A Test Procedure is a set of detailed instructions for setting up, executing, and evaluating the results of a given Test Case.

The final output of the UIT and UIT_sd methodologies, is therefore a set of Test Procedures, derived exclusively by the UML documentation without requiring to introduction of additional formalisms. It is worth noting that, in the following sections, we detail the Test Case and Test Procedure derivation only in for the SDs because the application of the two methodologies for the CDs is nearly the same

5.5.1 Category Partition Method

The Category Partition (CP) is a well-known and quite intuitive method proposed in the late 1980's [OB88] to derive functional tests from the specifications written in structured, semiformal language.

CP provides a systematic, formalized approach to partition testing that is one standard functional testing methodology. Generally speaking, partition testing is based on the simple idea that the input domain is first divided into several equivalence classes (also called partitions, although in order to be true partitions these should be non-overlapping, which is rarely the case in practice); then one or few tests are selected from within each of the identified partitions, as representative of the behaviour of the entire class.

The first step of the CP method is to analyse the functional requirements in order to divide the analysed system into functional units that can be tested separately. A functional unit can be a high-level function or a procedure of the implemented system. For each identified functional unit, the tester identifies the environmental conditions (the required system properties for a certain functional unit) and the parameters (explicit inputs for the unit) that are relevant for testing purposes: these are called the categories. The test cases are then selected by taking the significant values of each category, which in CP are called the choices. A complete set of test cases is obtained by taking all possible combinations of choices for all the categories. To prevent meaningless combinations or pairs of contradictory choices, the categories can be annotated with constraints, e.g., in a test case a choice from one category cannot occur together with certain choices from other categories.

The CP method has been implemented by Siemens in the TDE tool, that automatically constructs the test cases from the specifications expressed in a

dedicated semi-formal specification language, called TSL. The CP method has encountered wide interest in the literature, and has inspired the further development of a large number of test methodologies, also using formal languages such as Z [SP99].

5.5.2 UIT_sd

UIT_sd, similarly to the UIT method, constructs the Test Procedures using solely information retrieved from the UML diagrams. UIT_sd is an incremental test methodology; it can be used at diverse levels of design refinement, with a direct correspondence between the level of detail of the scenario descriptions and the expressiveness of the Test Procedures derived. All the SDs relative to a selected integration stage constitute the basis for the UIT_sd method. For each selected SD, the algorithm for Test Procedures generation is the following:

1. **Define *Messages_Sequences***. Observing the temporal order of the messages along the vertical dimension of the SD, a Messages_Sequence is defined considering each message with no predecessor association, plus, if any, all the messages belonging to its nested activation bounded from the focus of control region [UML]. A Messages_Sequence represents a behaviour to be tested and describes the interactions among objects necessary for realizing the corresponding functionality.
2. **Analyse possible subcases**: the messages involved in a derived Messages_Sequence may contain some feasibility conditions (e.g., if/else conditions). These conditions are usually described in the message notes or in the message specification and are formally expressed using the OCL notation [WK99]. If these feasibility conditions exist, a Messages_Sequence is divided into subcases, corresponding to the different possible choices.
3. **Identify *Settings Categories:*** for each resulting Messages_Sequence, we define the Settings Categories as the values or data structures that can influence its execution. In detail, they can be determined:
   - From all the messages involved, by considering their input parameters;
   - From the analysis of possible Class Diagrams to which the messages belong, by examining the attributes and data structures that can affect the observed interactions.
4. **Determine *Choices:*** for each Settings Category and for each Message belonging to a Messages_Sequence, the possible choices are identified as follows:

- for the Messages, they represent the list of specific situations, or relevant cases in which the messages can occur;
- for the Settings Categories, they are the set or range of input data that parameters or data structures can assume.

5. **Determine *Constraints* among choices:** the values of different choices in a Messages_Sequence may turn out to be either meaningless or even contradictory. To avoid this, the Category Partition methodology suggests introducing constraints among choices. These are specified by assigning to choices certain *Properties* used to check the compatibility with other choices belonging to the same Messages_Sequence, and by introducing the *IF Selectors*, which are conjunctions of previously assigned properties.
6. **Derive *Test Procedures***: a Test Procedure is automatically generated for every possible combination of choices, for each category and message involved in a Messages_Sequence. For each analysed SD, a document, called the *Test Suite*, collects all the derived meaningful Test Procedures grouped by Messages_Sequences.

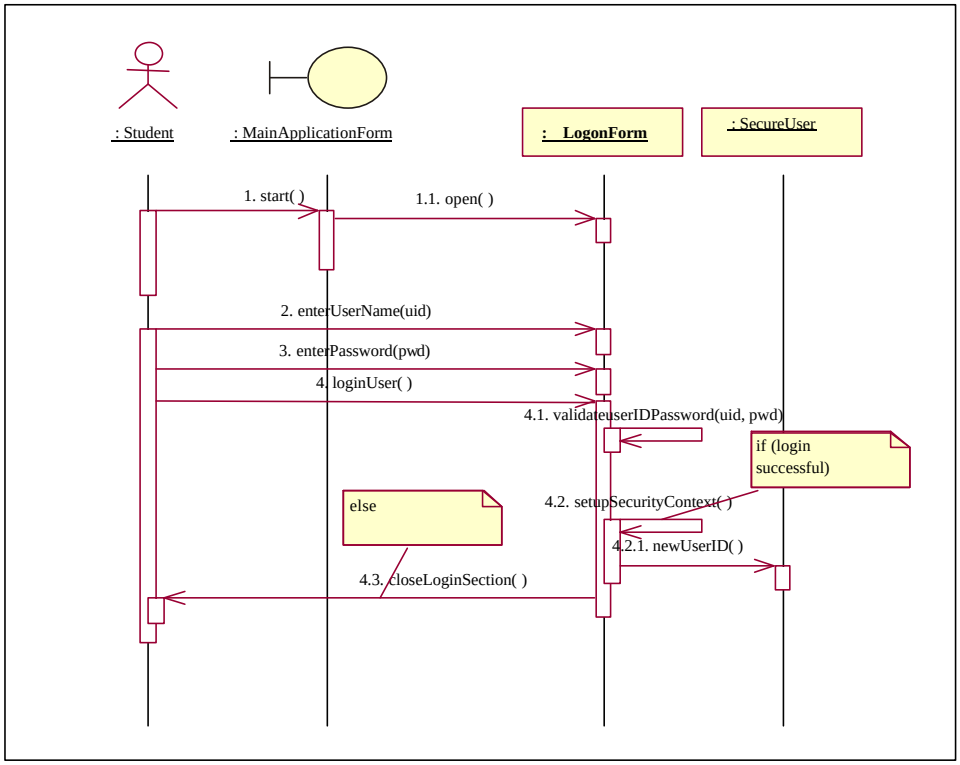


Figure 3 Sequence Diagram "Login-Main Flow" from CRS example of Section 5.7.1

Below, we report an example of UIT_sd application to the SD `Login-Main Flow` of Figure 3. Following the sequencing of messages along the vertical axis it is possible to initially define (Step1) four Messages_Sequences (M_S) such as:

- M_S1: 1.start()
  1.1.open()
- M_S2: 2.enterUserName(String)
- M_S3: 3.enterPassword(String)
- M_S4: 4.loginUser(),
  4.1validateUserIDPassword(String, String),
  4.2.setupSecurityContext(),
  4.2.1.newUserID(),
  4.3.closeLoginSection()

As described in Step 2, a feasibility condition in messages 4.2 and 4.3 can be observed: the value of `login successful` determines the execution of messages 4.2.1 or 4.3 so that Messages_Sequence 4 is split into two different subcases:

- M_S4.1: 4.loginUser(),
  4.1.validateIDPassword(String, String),
  4.2.setupSecurityContext(),
  4.2.1.newUserID()
- M_S4.2: 4.loginUser(),
  4.1.validateIDPassword(String, String),
  4.3.closeLoginSection()

For each derived Messages_Sequence, the Settings Categories can be identified (Step 3). In M_S4.1, for example, the categories are: `uid` and `pwd`, representing the parameters of the messages involved. Then for each message and for each Settings Category it is necessary to determine the Choices(Step 4). Figure 4 shows the definition of Choices for M_S4.1 and the Constraints values (Step 5) associated to the Choices in square brackets. Finally, as described in Step 6, the relevant Test Procedures are generated; the fixed amount of Test Procedures (as imposed by the strategy application) is randomly extracted from the potentially derivable ones. In Figure 8 we report, for the SD `Login-Main Flow`, the Messages_Sequences and the Test Procedures as derived by the tool Cow_Suite. A detail of one of the latter is shown in Figure 5 .

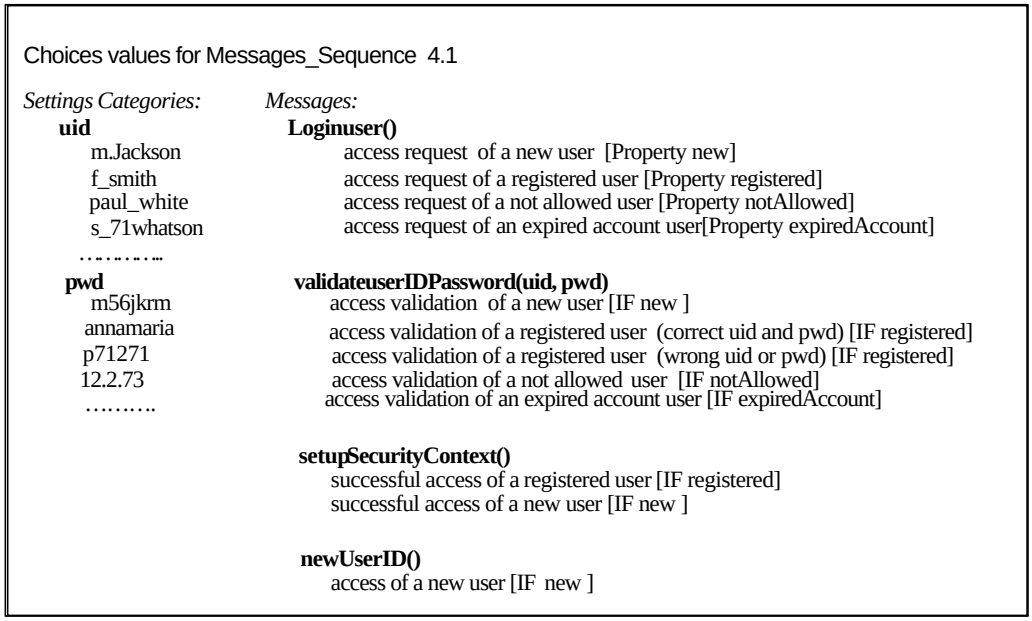


Figure 4 Choices values for Messages_Sequence 4.1

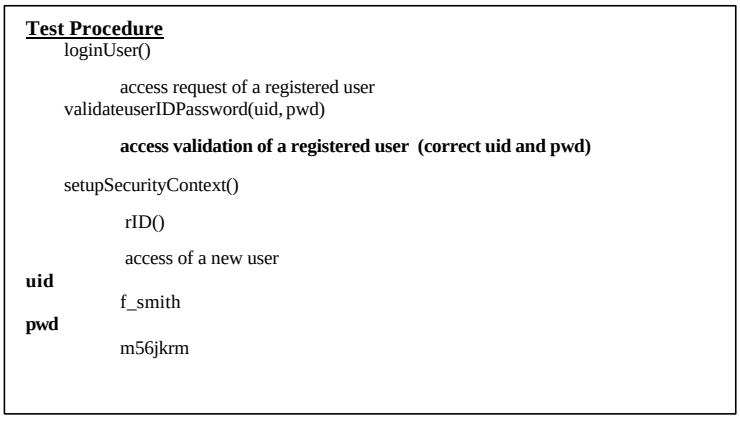


Figure 5 Test Procedure example

5.6 Cow_Suite Tool

The Cow_Suite approach can be naturally adopted and automated by industries using any UML design tool. We have implemented it in the Cow_Suite tool, in particular designed to be compatible with Rational Rose [RRT], one of the most widely used commercial tools for UML design. The existing Cow_Suite version retrieves the information extracted by Rose from the UML design using the REI (Rational Rose Extensibility Interface) libraries.

The Cow_Suite tool consists of three working windows: Cowtest, UIT and Test Specification, implementing respectively the Cowtest approach, the methodology UIT_sd and the Test Procedures generation.

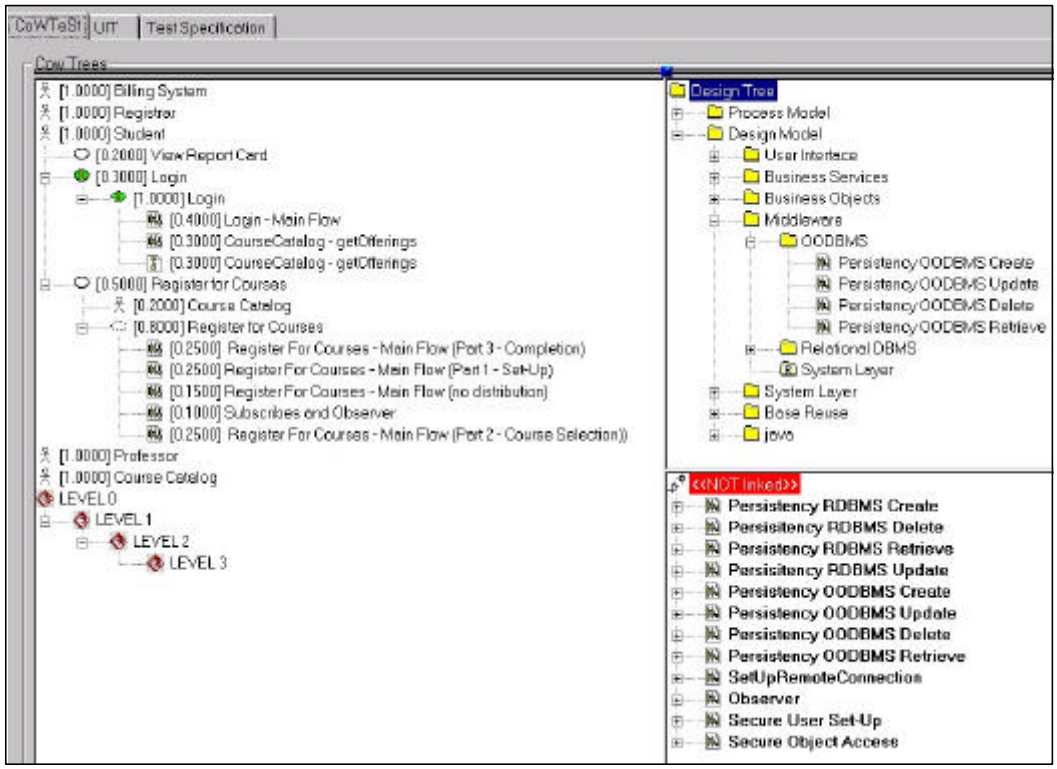


Figure 6 Main Cow_Suite tool window with Main Trees, Design Trees and Not Linked elements

The execution starts by analysing the Rose *.mdl* file (the internal representation of the parsed UML diagrams) and proceeds with the construction of the Main Trees and the Design Tree. Figure 6 shows a representation of the Main Trees, the Design Tree and the list of "Not linked" elements[2]. Observing this figure, we notice that , as the design evolves, the tool continuously provides a complete overview of the specification status of the diverse system functionalities.

Considering every Main Tree, the tool, by default, distributes the weights in a uniform way (Sec. 5.4.2.1) among the nodes at the same level of integration. However the user can always modify any of the assigned weights, and the values of the other nodes are automatically normalized. Following the step described in Sec.

[2] This figure, likes the others in this section, refers to the case study that will be presented in the next section

5.4.2.2, the user selects an integration level on a Main Tree and directly chooses the test strategy to use in a dialog window. For each selected integration stage, the tool directly derives a weighted subtree according to the chosen test criterion. In Figure 7, an example of the UIT Tree is reported. In particular the SD nodes keep track of the number of Test Procedures that must be developed according to the test strategy selected. In Figure 7, the left window shows the selected subtree, while, on the top right, all the SDs are collected together. In the bottom right window only the user selected SDs are listed.

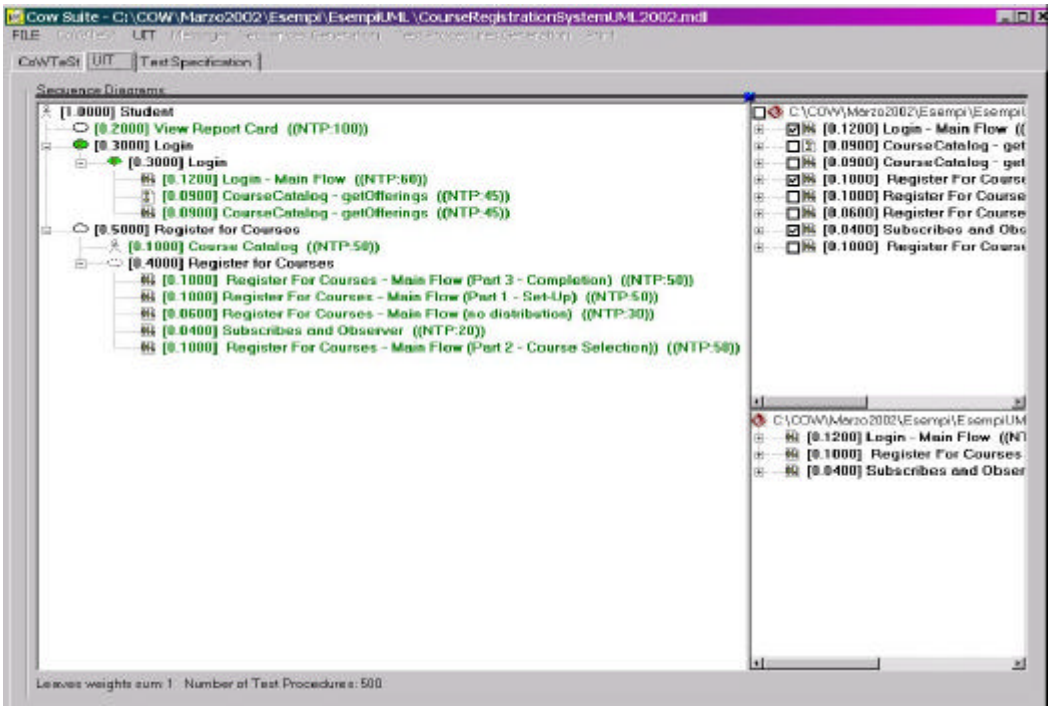


Figure 7 UIT Window with the derived UIT Tree, the set all SDs found and the selected SDs.

Then, for each selected SDs, the Cow_Suite tool automatically constructs the Messages_Sequences applying UIT_sd. On the left Figure 8 shows an example of a list of Messages_Sequences. Each Messages_Sequence contains the lists of all Messages and the Settings Categories involved plus its feasibility condition (where existing).

After Messages_Sequences derivation, the user, using several dialogue windows, can interact with the tool for inserting the Choices values, after which the Test Procedures are automatically derived. As explained in Section 5.5.2, the tool automatically excludes the combinations of parameters that result contradictory or

meaningless. On the right Figure 8 shows some of the final resulting Test Procedures.

So far the Test Suite document is a text file document, but the Test Procedures final format can be easily adapted to become the input format of a particular Test Driver. In this regard, we remark that the Cow-Suite tool does not execute the derived Test Procedures: for this purpose Cow_Suite should interact with a test driver, to which the derived tests should be passed to be automatically launched.

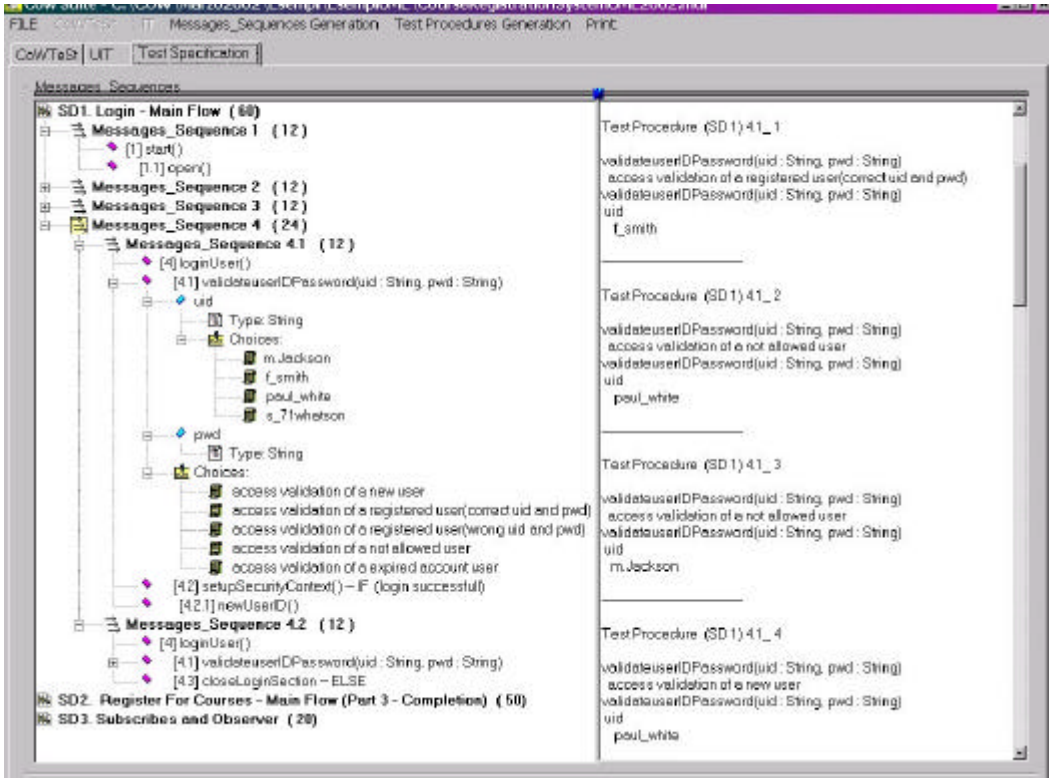


Figure 8 Messages_Sequences, Choices and Test Procedures for the SD Login-Main Flow

5.7 Applying Cow_Suite to Course Registration System

In this section we present the application of the Cow_Suite methodology to a case study, the Course Registration System [CRS]. In particular we discuss the Cowtest application and the integration with this strategy with the UIT methodology.

5.7.1 Course Registration System

The Course Registration System (CRS) is a fictional project to develop an on-line course registration system for Wylie College, and is well-known in the UML

literature [CRS]. The project is intended to replace the existing procedure for courses registration, which is based mainly on the personal interaction of the registrar with the students and professors and only support access through the clerk in the Registration Office. The new system will therefore enable professors and students to access the system using PCs connected to the Wylie College computer network and by any personal computer connected via the Internet.

The main problem with the existing system was the limited flexibility in the procedure followed by students for registering to the courses. They had to complete a course registration form and submit it to the registrar, who took up to 2 weeks just to examine the form and another week to send the confirmation back to the students. Instead, with the new system the CRS users, students, professors and a registrar can access the system via a login function through PC clients and quickly find the required information like the course availability and assignment. Briefly the main requirements for the new system will be:

- A student can either register for courses belonging to the current semester course catalogue or view his/her own data relative to the previous semester.
- A professor can select the courses he/she wants to teach from the course catalogue, also defining the dates and times the specific course will be given, and submit the grades.
- A registrar is in charge of professors and students' information. He/she maintains and verifies the data and course registrations, checking that there are enough people per course, and notifies the students in case the required courses are cancelled.

The CRS also interfaces with two existing system: the Billing system, which keeps track of each registered student in each course offering that is not cancelled, so the students can be billed, and the Course Catalog System that represent the database of the course information.

Typically in real project development not all the system functionalities are developed contemporaneously or are specified at the same level of details. This is the situation we consider as well. We assume that the software developer concentrates first on the realization of the student system interaction, represented by the system functionalities called: *Login*, used by the students to log into Course Registration System; *View Report Card*, that allows the students to consult their report cards for the previously completed semester; and finally *Register for Courses*, that allows the students to register to courses in the current semester. In particular the Course

Catalog System provides a list of all the course offerings for the current semester, so that the students can also modify or delete previous course selections, if the changes are made within the add/drop period at the beginning of the semester.

5.7.2 Cowtest Application

The Cowtest application begins with the analysis of the UML documentation available fro the CRS case study. In particular, as described in Section 5.4.1 the first representation of the Main Graph is derived by using the information of the Use Case View. We show in Figure 9 the Main Graph obtained. In this example, the UCs are quite simple not further refined into sub-UCs, therefore only a level of UCs is derived.

This graph is then integrated, as described in Section 5.4.1.2, with the information of Logical View when it is available. In Figure 10 we report the upgraded Main Graph in which both the information of the Use Case View and the Logical View are integrated. The UCs with a dotted edge represent the use case realizations.

It is worth noting that the CRS is a fairly simple example, which nearly has a UML project specification complete in every part. As result the Main Graph obtained is connected but generally this is not a typical situation.

As discussed in Section 5.4.1.2, once the Main and the Design Graphs were completed, it was not possible, only using the information of the UML specification, to link the data of the former with those of the latter. Specifically, considering for instance the Login function of the Main Graph, it was very hard to individuate in the Design Graph the packages that realized this functionality. To overcome this problem, even if we didn't know the real association between UCs and packages, we tried to derive a probable and sufficiently realistic Design Link at least for the Login function by consulting further documentation. The obtained result is shown in Figure 11. As explained in Section 5.4.1.2 the Design link collects the list of the packages of the Logical View that implement the UC associated to the use case realization considered.

The derived graphs are then used for the application of the DFS_Mod algorithm, as described in Section 5.4.1.3, to obtain the Main Trees and the Design Trees. To this purpose, considering the Main Graph shown in Figure 10, we report in Figure 6 the structure of the Main Tree rooted at the actor Student. This tree is only focused on the interaction of the system with this actor, excluding all the other system

functionalities not directly involved, because the purpose is to test this collaboration separately.

As shown by the figure the tree presents some marked parts due to the presence in the Main Graph of nodes connected to more than one design elements: For instance the UC node `Login` at first level is filled (and labelled with a "R" not visible in the figure), because it is a multiply used functionality, i.e. more than one actor is associated to it, as shown in the Main Graph.

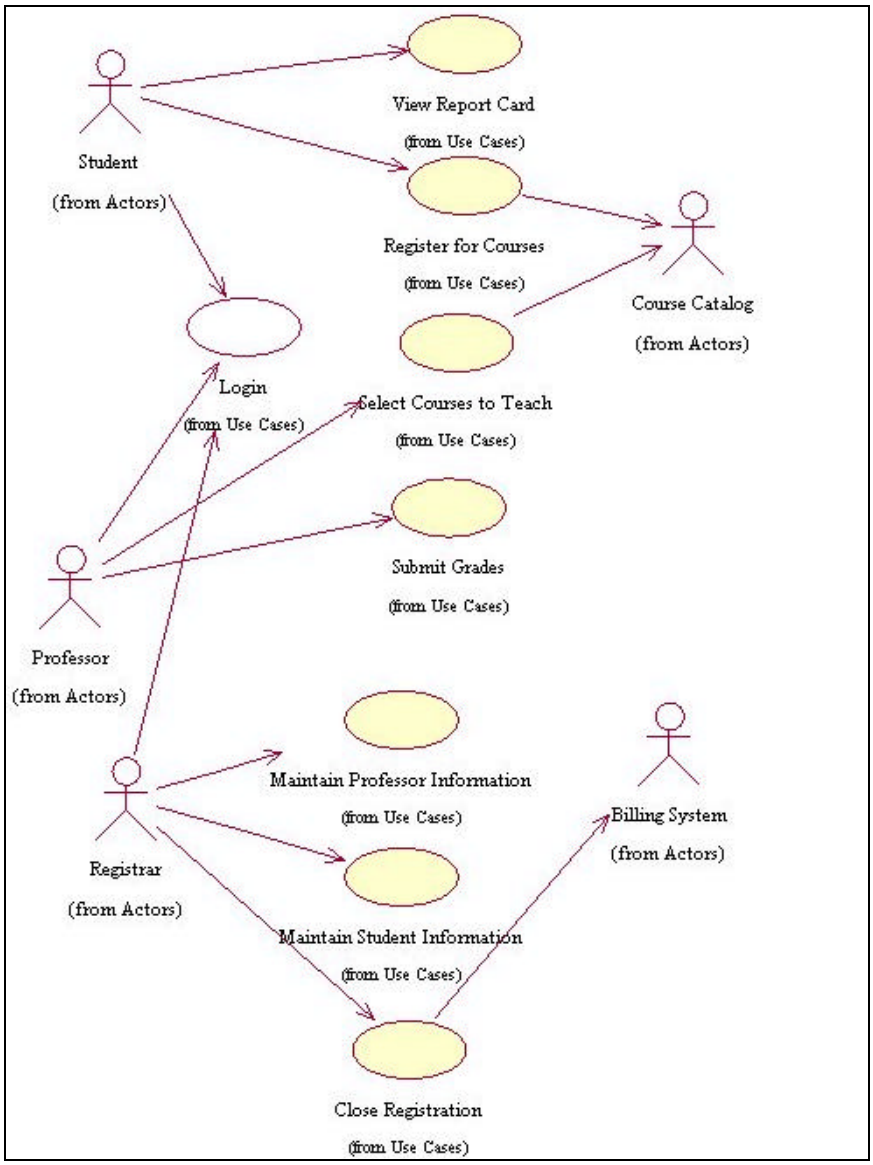


Figure 9 The Main Graph after the analysis of the Use Case View Data

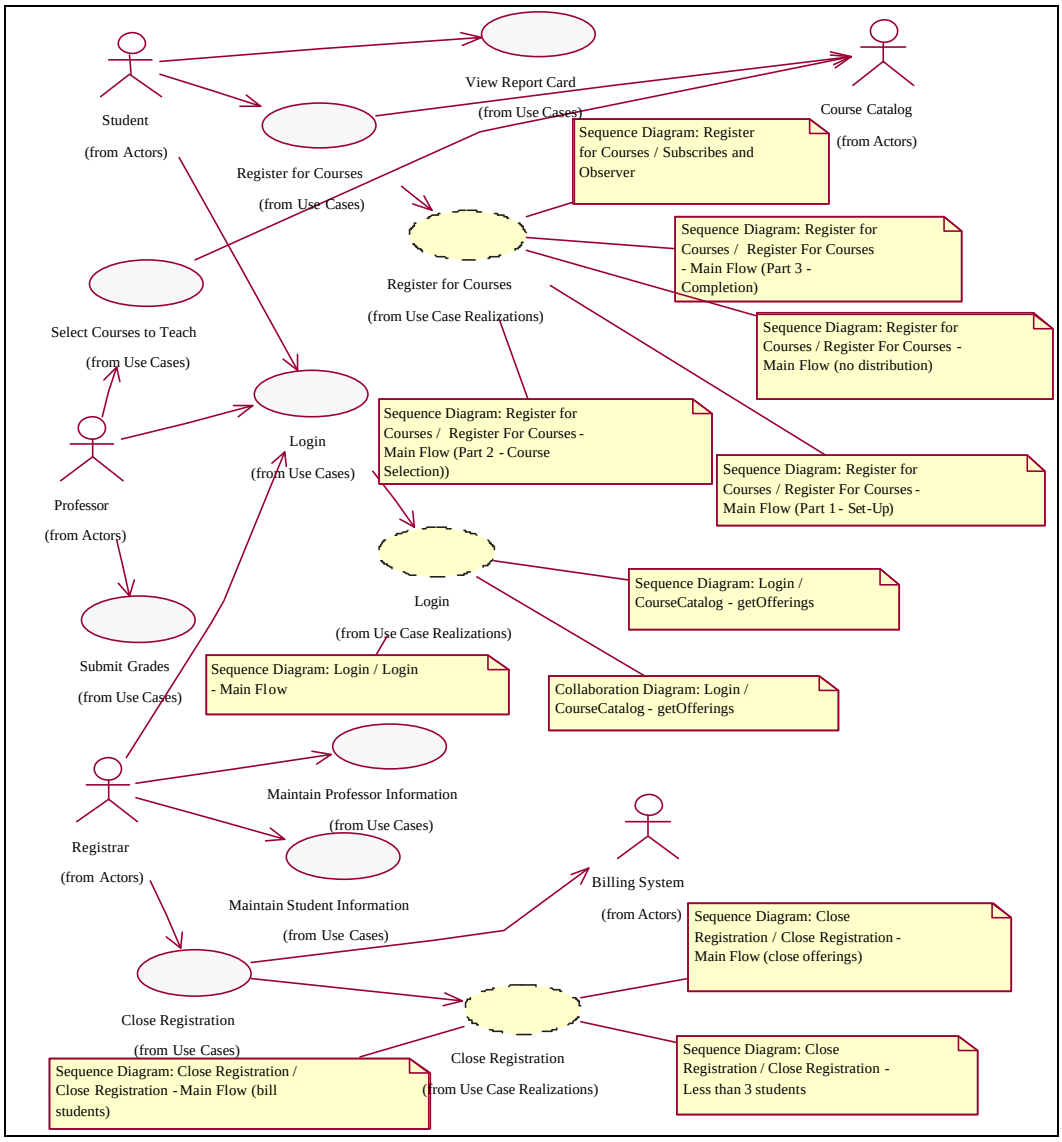


Figure 10 Main Graph of the Course Registration System

Once the Main Trees have been defined it is necessary to assign the importance values to each node as described in section 5.4.2. To this purpose, considering the CRS case study, we assume, for example, that during its development, the `Login` and `Register for Courses` must be completely defined and implemented, while `View Report Card` is an already developed functionality. It is important to specify that this is just one of the possible cases considered to show a criterion to assigning the weights, it is not the real situation encountered during the CRS development.

Consequently in our example `Login` and `Register for Courses` are new system functionalities, and therefore we assigned to them a greater weight than that associated to the already built `View Report Card`. In particular `Register for Courses` is more complex, in term of implemented features, than `Login`, so its testing must be more accurate. Based on these considerations we assign the values 0.50, 0.30, 0.20 to `Register for Courses`, `Login` and `View Report Card,` respectively. In Figure 6, the weights assigned to each node are represented by the numbers reported in square brackets close to the node name.

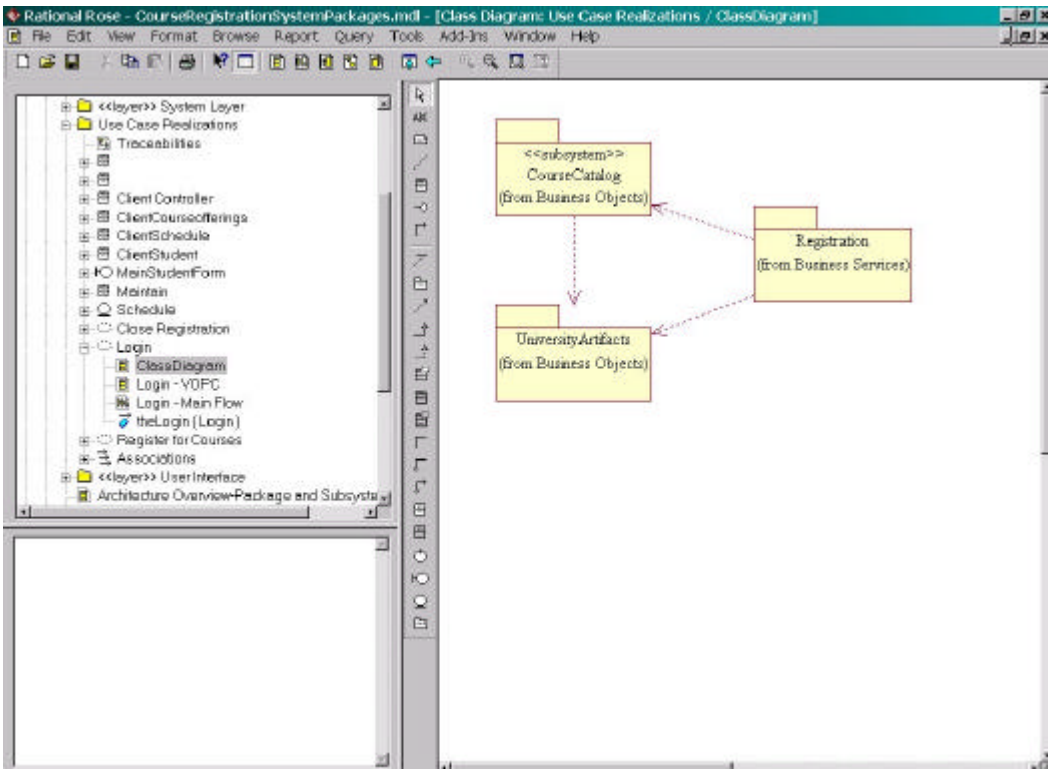


Figure 11 Design Link Package for the Login function

Then the nodes at lower levels are considered from time to time and the weights assigned. For example, considering Figure 6, for the node labelled `"Login"` at the second level, that is the use case realization of the use case `"Login"` at level one, the weights assignment of its children is: 0.4 to the `Login-Main Flow` because it is the SD representing the main behaviour of the login functionality, 0.3 to the others because they represent the minor interactions between the objects involved.

The weights assignment is then used to derive the final weight of each node after the integration stage selection, as described in Section 5.4.2.2. To this purpose we

report in Table 1, considering the tree rooted in the Student node of the CRS example, the nodes that belonging to the different integration stages and in Table 2 the corresponding final weights when the 2nd integration stage is selected. In particular Table 2 is organized in the following manner: the first and second columns hold respectively the integration stage, and the names of all tree leaves considered. The third column shows the leaves critical profile, i.e., the importance values of each node visualized as the weights in square brackets in Figure 6. For example, the final weight of the SD `Register For Courses – Main Flow (Part 1 – Set-up)` is calculated as 0.1 = 0.25*0.8*0.5.

Integration stage	Tree nodes
1st int. stage	View Report Card, Login, Register for Courses
2nd int. stage	Login, Login–Main Flow, CourseCatalog–getOffering, CourseCatalog–getOffering, Course Catalog, Register for Courses, Register For Courses-Main Flow (Part 3 Completion), Register For Courses-Main Flow (Part1 Set-Up), Register For Courses-Main Flow (no distribution), Subscriber and Observer, Register for Courses-Main Flow (Part 2 Course Selection)
3rd int. stage	Login–Main Flow, CourseCatalog–getOffering, Register For Courses-Main Flow (Part 3 Completion), Register For Courses-Main Flow (Part1 Set-Up), Register For Courses-Main Flow (no distribution), Subscriber and Observer, Register for Courses-Main Flow (Part 2 Course Selection)

Table 1 Integration stages

In Figure 7 we report the final weights of the tree rooted in the Student as derived by the Cow_Suite tool.

For example the assigned number of tests for `Register For Courses – Main Flow (Part 1 – Set-up)` is given by $50 = \lfloor 500*0.1+0.5 \rfloor$.

Instead, taking into account the second test strategy proposed, i.e. Cowtest_ing with fixed functional coverage, the final weight of every leaf can be used to select among them those on which concentrate the test effort. Referring to the CRS example in Table 3we report in details some results obtained considering the second integration stage and several coverage degrees.

The table is organized in the following manner: the first and second columns hold respectively the names of all the tree leaves and the their relative weights at the second integration stage. The remaining columns are divided into two parts showing, respectively, the normalized final weight, *nwf*, and the minimum number of tests with respect to the fixed coverage percentage.

Integration Stage	Leaves names	**Critical profile**	**2nd Stage/NTest**	
1st Stage	View Report Card	0.2	0.2	100
	Login	0.3		
	Register for Courses	0.5		
2nd Stage	Login	1		
	Login–Main Flow	0.4	0.12	60
	CourseCatalog–getOffering	0.3	0.09	45
	CourseCatalog–getOffering	0.3	0.09	45
	Course Catalog	*0.2*	*0.1*	*50*
	Register for Courses	*0.8*		
	Register For Courses-Main Flow (Part 3 Completion)	0.25	0.1	50
	Register For Courses-Main Flow (Part1 Set-Up)	*0.25*	*0.1*	*50*
	Register For Courses-Main Flow (no distribution)	0.15	0.06	30
	Subscriber and Observer	0.1	0.04	20
	Register for Courses-Main Flow (Part 2 Course Selection)	*0.25*	*0.1*	*50*

Table 2 Test cases distribution at different integration stages

Leaves names	**2nd Stage weights**	**70%coverage *nwf* /NTest**		**80%coverage/ *nwf* /NTest**		**90%coverage/ *nwf* /Ntest**		**100%coverage/ *nwf* /NTest**	
View Report Card	0.2	0.27	2	0.2469	2	0.2222	2	0.2	5
Login–Main Flow	0.12	0.1667	1	0.1481	1	0.1333	1	0.12	3
Course Catalog	0.1	0.1389	1	0.1235	1	0.1111	1	0.1	2
Register For Courses-Main Flow (Part 3 Completion)	0.1	0.1667	1	0.1235	1	0.1111	1	0.1	2
Register For Courses-Main Flow (Part1 Set-Up)	0.1	0.1389	1	0.1235	1	0.1111	1	0.1	2
Register for Courses-Main Flow (Part 2 Course Selection)	0.1	0.1389	1	0.1235	1	0.1111	1	0.1	2
CourseCatalog–getOffering	0.09			0.1111	1	0.1	1	0.09	2
CourseCatalog–getOffering	0.09					0.1	1	0.09	2
Register For Courses-Main Flow (no distribution	0.06							0.06	1
Subscriber and Observer	0.04							0.04	1

Table 3 Leaves selection on the base at different values of functional coverage

In this case considering Table 3, if we wish to cover 80% of the functionalities available it is sufficient to include the nodes: `View Report Card`, `Login- Main Flow`, `Course Catalogue–getOffering`, `Course Catalog`, `Register for`

Courses, Register For Course–Main Flow (Part 3-Completion), Register For Courses–Main Flow (Part 1 Set-Up), Register For Courses–Main Flow (Part 2–Course Selection). The sum of their final weights times 100 is equal to 81. Moreover, using the final weights of the selected leaves, normalized so that their sum is still equal to 1, it is also possible to derive the minimum number of test cases required to reach the fixed coverage. In this case the minimum number of test cases is 8, one test per leaf except View Report Card which required 2 tests cases.

5.7.3 Combining UIT_sd and Cowtest

We do not describe here the Test Cases and Test Procedures derivation by applying the UIT_sd methodology, since it has been already described in Section 5.5.2. For this purpose, some of the Test Procedures derived form SD Login-Main Flow of the tree rooted in the Actor Student, are shown in Figure 8. Here we mainly concentrate on explaining the integration of the Cowtest with the UIT methodology and for this purpose we slightly modified the UML design of the quite simple CRS case study.

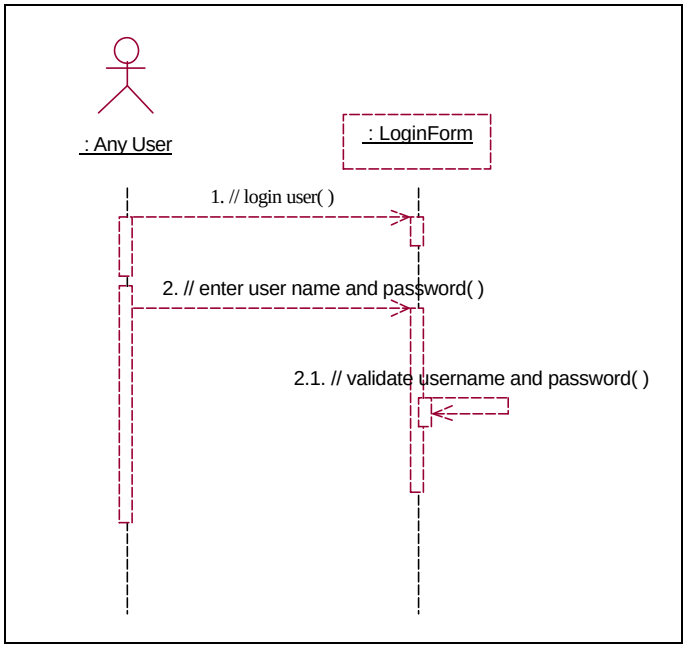


Figure 12 SD labelled Login – BasicFlow

Generally from the UML designs of the real world projects more complex trees with a higher number elements can be derived. In these cases the differences between

the test cases, derived from the SDs associated to diverse integration stages, become more evident with respect to CRS, as well as the relation between integration stage and level of integration testing exercised.

The changes of the UML documentation consist of linking to the UC labelled Login of the Use Case View, the high level SD called Login–BasicFlow reported in Figure 12. This is a very simple SD, which describes the functionality represented by the UC.

Naturally the addition of this SD modifies the structure of the derived Main Trees. Figure 13 shows the changes relative to the tree rooted in the actor Student of the CRS. It is beyond the scope of this section to discussion about the new assignment of the weights to the nodes; we therefore adopted the uniform distribution.

As discussed in section 5.4.2.2 the selection of an integration stage corresponds to determining the amount of information to use for integration testing and hence the structure and granularity of the Test Cases derived applying UIT_sd.

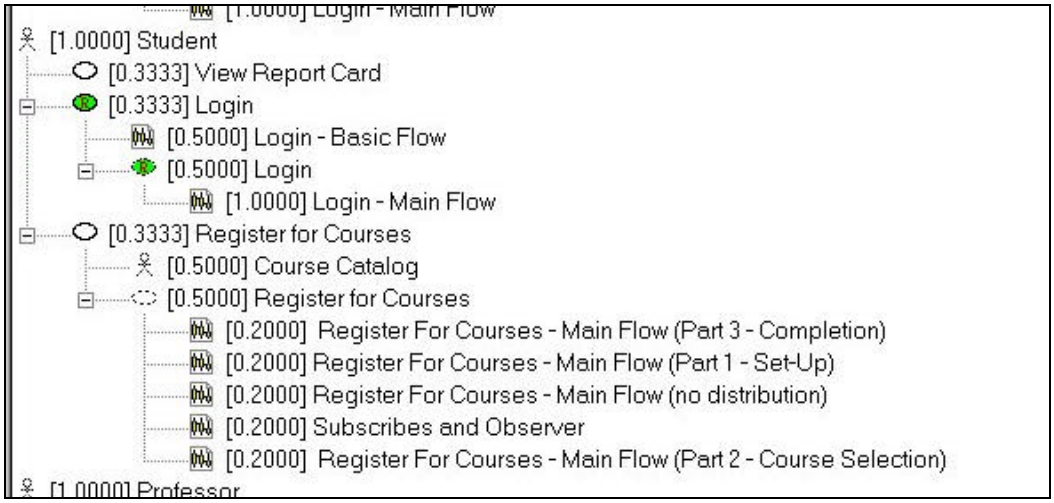


Figure 13 The tree rooted in the Actor Student modified with the addition of Login-Basic Flow

Specifically, assuming the adoption Cowtest_ing with a fixed number of test cases, if in the above tree the first integration stage is selected only the newly added SD (Login-BasicFlow) will be considered for the UIT_sd application. Figure 14 shows the set of derived Test Cases derived applying the Cow_Suite tool. Observing their structure, it is evident that the Test Cases only have the purpose of verifying the correctness of interactions among the components that will realize the functionality

described in the UC Login. The degree of detail is voluntarily high and reflects the granularity of information of the selected integration stage.

Sequence Diagram "Login – Basic Flow"	
Test Case 1 *Description:* *Precondition:* *Flow of Event (Messages_Sequence):* loginUser *Categories:* *Settings Categories:* Users DataBase *Interactions Categories:* LoginUser *Post Condition:* *Comment*:	**Test Case 2** *Description:* *Precondition:* *Flow of Event(Messages_Sequence):* Enter user name and password Validate user name and password *Categories:* *Settings Categories:* UserNames Database Passwords and User Name Lists *Interactions Categories:* Enter user name and password Validate user name and password *Post Condition:* *Comment*:

Figure 14 Test Cases derived by the SD Login-Basic Flow

If instead the second integration stage is selected, assuming again adoption of the Cowtest_ing with a fixed number of test cases, the SD Login –Main Flow is also selected (Figure 3). It represents the description of all the necessary operations (i.e. messages that the different objects exchange with each other) to implement the functionality described in the UC Login. In this case Figure 15 shows the set of Test Cases derived with the Cow_Suite tool. Comparing them with the Test Cases derived from the SD Login–Basic Flow several differences are revealed:

- The objects involved are detailed at various levels of description. In the SD Login – Basic Flow the LoginForm is only the high level description of a system component that will be realized by the objects of the SD Login–Basic Flow.
- The operations in the Test Cases derived from the SD Login–Basic Flow are more detailed, focusing on implementation and specific for the integration test at low level.
- In both cases the Test Cases structure depends only on the Messages_Sequences individuated in the SDs. Therefore, it is not possible to relate a Test Cases of the SD Login – Basic Flow to one of the SD Login – Basic Flow.

As revealed by this simple example, the higher the integration stage selected for applying the different test strategy the more detailed is the level of integration verified by the derived Test Cases.

<table>
<tr><td colspan="3">Sequence Diagram ”Login – Main Flow”</td></tr>
<tr><td>Test Case 1

Description:
Precondition:
Flow of Event
start()
open()
Categories:
Settings Categories:

Interactions Categories:
Start()
Open()
Post Condition:
Comment :</td><td>Test Case 2

Description:
Precondition:
Flow of Event:
EnterUserName()

Categories:
Settings Categories:
uid
Interactions Categories:
EnterUserName

Post Condition:
Comment:</td><td>Test Case 3

Description:
Precondition:
Flow of Event:
EnterPassword()

Categories:
Settings Categories:
pwd
Interactions Categories:
Start()
Open()
Post Condition:
Comment:</td></tr>
<tr><td colspan="2">Test Case 4.1
Description:
Precondition:
IF (login successful)
Flow of Event:
loginUser()
validateuserIDPassword(uid, pwd)
setupSecurityContext()
newUserID
Categories:
Settings Categories:
uid
pwd
Passwords and UserNames Database
Interactions Categories:
LoginUser
ValidateuserIDPassword
SetupSecurityContext
NewUserID
Post Condition:

Comment:</td><td>Test Case 4.2
Description:
Precondition:
ELSE
Flow of Event:
loginUser()
validateuserIDPassword(uid, pwd)
closeLoginSection

Categories:
Settings Categories:
uid
pwd
Passwords and UserNames Database
Interactions Categories:
LoginUser
ValidateuserIDPassword
CloseLoginSection

Post Condition:

Comment:</td></tr>
</table>

Figure 15 Test Cases derived by the SD Login – Main Flow

5.8 Comparing Manual vs. Automated Test Case Derivation.

As observed in the preface of this Chapter, Cow_Suite can be applied during the development process either combined with Propean approach (Chapter 4) to derive parameters of interest, or in isolation to derive a meaningful Test Plan. In this section we focused in the latter aspect, describing the application of Cow_Suite to a real-world case study provided by Ericsson Lab Italy (ERI) [BIL03]. Using our tool we derived a detailed test case plan, called *UIT test plan* for the Integration Testing of some new functionalities to be added to an existing system. The UIT test plan was automatically derived outside the production processes, exclusively using the UML diagrams developed during the analysis and design phases. The ERI personnel had independently derived another test plan (the "official" one), called *ERI test plan*, for the same functionalities. The ERI test plan was developed manually, following standard in-house procedures at ERI and was based mainly on the personal expertise of the people involved and their knowledge of the system. In the following sections we report the description of the case study and two test plans (Sections 5.8.1, 5.8.2, 5.8.3) and their qualitative and quantitative analysis (Section 5.8.4).

5.8.1 Case Study

The case study concerned a project whose aim was to develop an IP Telephony system to support GSM communications based on H.323 architecture [H323]. Although the system was not wholly under ERI responsibility, a significant number of independent subsystems were managed and developed in-house by ERI. The subsystems under ERI's control regarded mainly H.323 gatekeeper functions implemented by the Sitekeeper and User Agent subsystems. The Figure 16 depicts the entire system, together with the subsystems under ERI responsibility SK, UA, H.323 proxy and the associated plug-in.

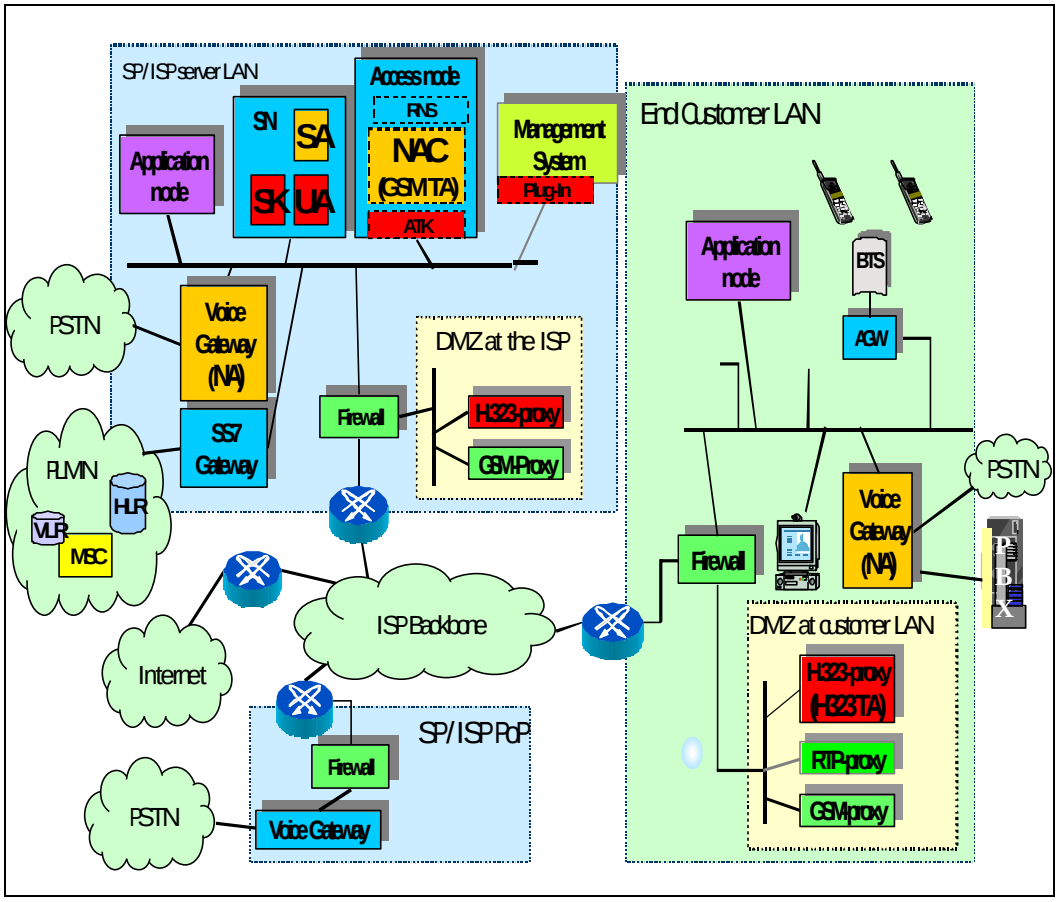


Figure 16 Description of the system

Table 4 provides a brief description of some of the system components shown in the above figure.

The specific feature used to compare ERI's manual vs. UIT-based automated test-case definition was the Basic Routing Enhancements (BRE).

This feature represents an improvement of the routing functionality in the GSM on the Net system, which is a new multimedia system based on IP protocol The upgrade mainly regards extension of some tables through the addition of new parameters, the implementation of new functionalities for determining the enterprise or the User Agent Group associated, giving a certain number.

As proper implementation of BRE implies modification of the Site Keeper, SK, and User Agent, UA, an accurate and specific test plan was needed.

BTS	the usual Base Station for GSM radio transmission
A - Bis Gateway (AGW)	the component that packs the GSM messages into IP packet, to allow transport over the IP network
Terminal Agent (TA)	the component that allows all terminals H.323 compliant to access the system
Access Node	a control node for the GSM terminals. It is composed of: **Radio Network Server (RNS):** provides the radio network management and the traffic functions **Network Access Controller (NAC):** a Terminal Agent for the GSM terminals. In th e NAC the system provides the protocol translation between GSM proprietary protocols
Service Node (SN)	is the main component of the system. The SN is composed of: **User Agent (UA):** in the UA are implemented all the users system functionalities. In particular: User **Service Agent (SA):** in the SA there the implementations of supplementary services, such asVirtual Private Network, Calling Line Identity Presentation/Restriction, Call Forwarding. **SiteKeeper (SK):** in the SK is the interface between the SN and the system access provider. All terminals (if necessary converted to H.323 by an access agent) enter the SN through the SK. The SK performs the routing for calls and resources management.
Manageme nt System	the component for efficient management of the system
SS7 Gateway	the gateway dedicated to PLMN interface

Table 4 Description of components

5.8.2 ERI Test Strategy

To better illustrate the test strategy adopted by ERI, a brief description of the project's scope is in order. The project had to implement eleven features, which after careful analysis we discovered to be nearly all independent. Therefore, due to the project's short timeframe, a parallel development life cycle was adopted rather than

an incremental one. Although the features were independent, all components were affected by more than one feature. For this reason, the project involved identification of specific test strategies for each feature with the aim of covering the feature requirements as well as the architecture of the system as a whole.

Testing Activities	Characteristics	Responsability
Class Test	Executed both in static and dynamic mode	Design Team
Component Test WB	White Box, aiming at testing the interfaces among classes, described in a specific Test Plan	Design Team
Component Test BB	Black Box, aiming at testing in a simulated environment the functions implemented by the component and its behaviour, described in a specific Test Plan	Design Team
Node Test WB	White Box, aiming at testing the interfaces among components, described in a specific Test Plan	Design Team
Node Test BB	Black Box, aiming at testing in a simulated environment the functions implemented by the node and its behaviour, described in a specific Test Plan	Design Team
Feature Test Pre-Integration	Functional test in simulated environment using the real code, described in a specific Test Plan. The Test Plan is derived from the detailed requirements. The main purpose of the preintegration is to deliver to the Integration & Validation team a feature running and clean	Design Team
Feature Test	Functional executed in the target environment., described in a specific Test Plan. The Test Plan is derived both from detailed requirements and main requirements	Integration & Validation Team
Regression Test	Mandatory at the end of each feature delivery	I&V Team
Performance Test, Stability Test, Negative Test, Overload Test, Characteristic Test, Capacity Test	Described in a specific Test Plan	I&V Team

Table 5 Description of testing activities

The test strategy defined by the project comprises nine different testing activities as described in Table 5.

Nevertheless, not all the activities were mandatory; each feature had its own Test Strategy defining the test activities to be performed. The purpose of having a specific

test strategy for each feature was to arrive at the best trade-off between quality and time. In particular the BRE test strategy was to perform five different testing activities (the last common to all the features): Class Test; Feature Test Pre Integration; Feature Test; Regression Test (twice); Performance Test.

For the illustration purposes, herein we concentrate on the Pre-integration Test and present and compare the two different test plans with regard to this aspect of the BRE testing.

5.8.3 Test Plans Description

In this section we briefly describe the structure of two derived Pre Integration test plans. Specifically, in Section 5.8.3.1 we present the "official" test plan, (which we refer to as ERI_TP in the following), developed by ERI following standard in-house procedures, and in the Section 5.8.3.2, the UIT Test Plan, (henceforth referred to as UIT_TP), derived applying the Cow_Suite tool, based exclusively on UML-diagrams.

5.8.3.1 ERI Test Plan

The ERI_TP has been defined specifically for testing the BRE functionalities. Essentially, it is a natural language document describing the test cases configuration, as well as the test results expected in terms of checking that the BRE requirements have been fully covered. In drawing up the ERI_TP, the ERI staff bases their decisions solely on their personal knowledge, both for definition of the test cases and validation of their accuracy with the respect to the requirements.

The test plan was obtained wholly independently of the UIT_TP, without reference to the UML system description. Moreover, no tool or automatic device was applied for deriving the test cases. The test specifications were in fact defined "manually" according to the standard in-house procedures at ERI.

Once the test cases were defined, each test was then assigned to a specific test group representing high-level system functionality. The Project Manager uses such test groups to check requirement compliance.

In greater detail, each test case is divided into three separate parts: *Description*, *Precondition* and *Procedure*.

- The *Description* defines the goal of the test cases. Moreover, it provides a description of the environment, the entities involved in the test and the specific conditions under which the test should be run.

TEST GROUP 1: CALL ESTABLISHMENT WITH ENTERPRISE INFORMATION
This test group aims to verify that the Control Node is able to establish different kinds of calls using the Enterprise information.

TC 1: Basic call from internal user to External Network, Enterprise with public numbering plan, Enterprise determination based on e164 alias

Description
This test is made to verify that the typical call case from user to External Network works properly using the information of the Enterprise the user belongs to.
The needed Enterprise determination is performed using the e164 alias in the incoming SETUP message.

Precondition
- The file *MasterRoutes.def* must contain a row looking like this:
  1 (=UA) UAGname 1 (=e164 Route Type) EnterpriseName Digits 0
  An example could be: 1 UAGxxx 1 Ericsson 39067258 0
- The Enterprise (in our example "Ericsson") must be present in the file *Enterprise.def* .
- No Number Modification will be configured for the TA the calling user belongs to.
- A GW is to be added to the Network Topology; this Access Agent must be associated (via a proper Access Group) to a suitable route (let's make it for example "39068") and to the Enterprise the user belongs to. The file *MasterRoutes.def* could contain for example a row looking like this:
- 1 Agxxx 1 Ericsson 39068…
- Another GW is to be added to the Network Topology; this Access Agent must be associated (via a proper Access Group) to the same above route but to a different Enterprise (let's call it for example "Nokia"). The file MasterRoutes.def could contain for example a row looking like this:
- 1 Agyyy 1 Nokia 39068…

Procedure
Action:
Make a call from the user belonging to the first Enterprise (in our example "Ericsson") to the above GW. The first digits of the dialed number must match the above route (in our example a suitable Called Party Number could be "39068xxxx").
NB: please notice that the originating SETUP message MUST contain in the source address an e164 alias matching with the one defined in the table *MasterRoutes.def* (in our example the alias could be "390672580001").
Result:
The final result is the call termination towards the GW reserved for the Enterprise the user belongs to (in our example to the GW in the AG "AGxxx", not "AGyyy").
Comment:
Only the GWs reserved for the Enterprise can be used for routing calls.

Figure 17 ERI_TP Test case description

- The *Precondition* delineates the data structures involved in the test case. In particular, the values and the types of information they must contain are listed explicitly. Often, the precondition part also provides a natural language description of the behaviour that the test case must exhibit, the actions it is to perform and the conditions required for test execution.
- The *Procedure* part is in turn divided into three sections: *Action*, *Result* and *Comment*. The Action consists of a brief description of the steps necessary for

constructing the test case and assigning values to its variables. The Result section describes the expected outcomes of the test case. Finally, the Comment section may contain some notes or suggestions for proper execution of the test case.

In Figure 17 one of the ERI_TP developed test cases is reported.

5.8.3.2 UIT Test Plan

The UIT_TP is derived by applying the Cow_Suite tool to the UML description of the system. In particular, we distinguish two level of detail: the UIT_TP including only Test Cases or UIT_TP in which the Test Procedures are specified.

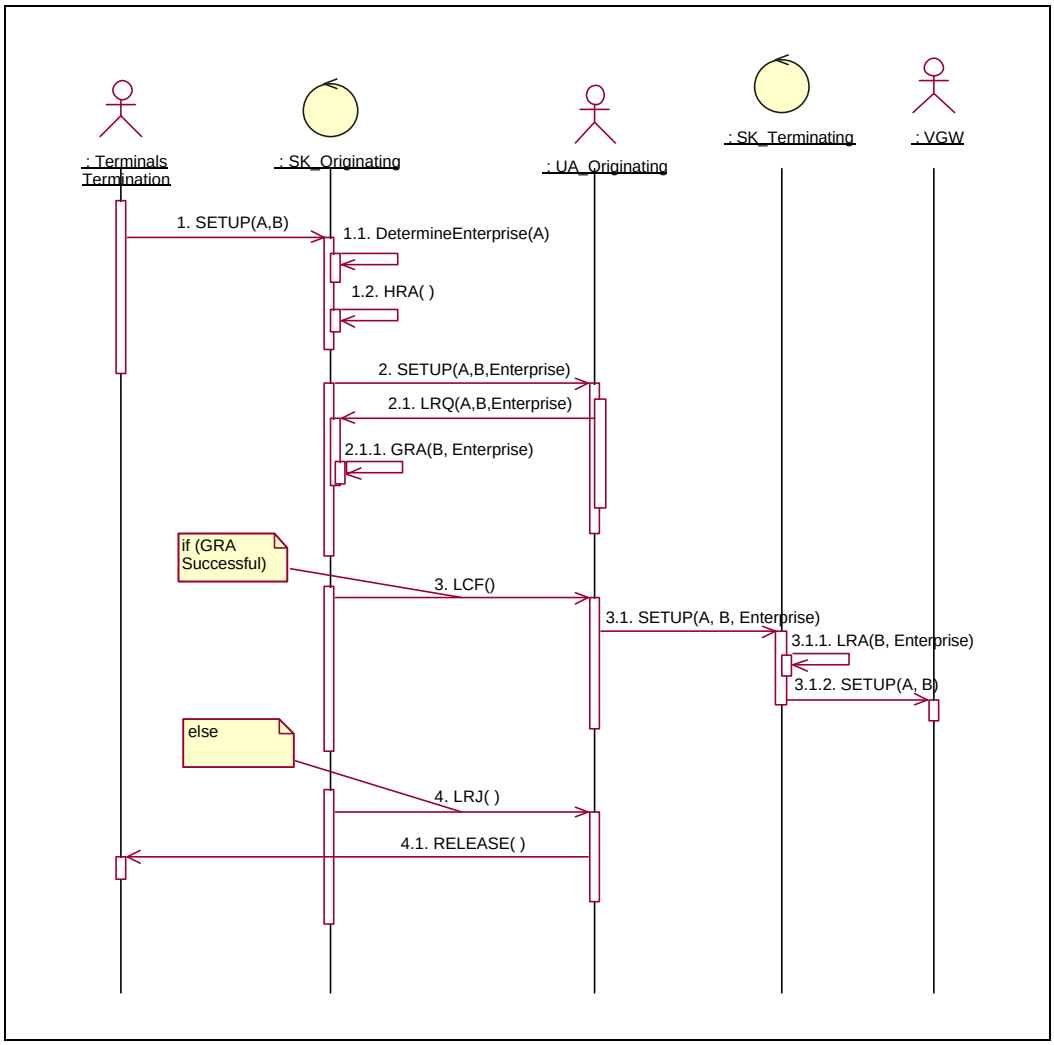


Figure 18 Sequence Diagram "Call user to External Network"

Depending on the degree of detail chosen, some evident differences result, regarding both the people involved in test plan specification, and the period of software development during which the UIT_TP can be derived.

Considering the UIT_TP at Test Cases level, it can be derived during the early stages of software development, long before the testing phase. Test Cases construction does not require any specific knowledge of the system, because it is derived automatically from the information in UML diagrams. In Figure 18 and Figure 19 respectively we show one of the SDs available and some of the individuated Setting and Interaction Categories. These data have been used for deriving the Test Cases of Figure 20.

Test Specification for "UA_Originating"

Settings

A

E164 phone number
email [*IF* Ericsson Enterprise]

B

E164 phone number

Enterprise

Ericsson [*Property* Ericsson Enterprise]
Nokia [*Property* Not (Ericsson Enterprise)]
Siemens [*Property* Not (Ericsson Enterprise)]
……

MasterRoute.def

array of MasterRoute records

Network Topology files

Interactions:

Setup(A, B, Enterprise)

call from a GON user to an external network when
a correct association in MasterRoute.def exists
call from a GON user to an external network when
a MasterRoute.def association does not exist

LCF ()

call from a GON user to an external network when
an association for B exists in MasterRoute.def

LRJ()

call from a GON user to an external network when
an association for B in MasterRoute.def does not exist

Figure 19 Settings and Interactions Categories Specification for UA_Originating

As shown by this figure each Test Case contains information useful for determining the interactions of the units involved and how to test them. Once derived, Test Cases are grouped into *Use Case Test Suites* (UCTS), which represent

the actions necessary in order to check correct performance of the functionality described in the UC. Figure 20 shows an example of UCTS corresponding to test group 1 of the ERI_TP reported in the Figure 17.

USE CASE TEST SUITE 1

Sequence Diagram "***BRE-Step1****: Call User to External Network; Originating Case/Terminating Case"*

Test Case 1
Description:
Precondition:
Flow of Event:
SETUP(A,B)
DetermineEnterprise(A)
HRA()
Categories:
Settings Categories:
A
B
MasterRoutes.def
Enterprises.def
Network Topology
Interactions Categories:
SETUP(..,..,..)
DetermineEnterprise(..)
HRA()
Post Condition:
Comment:

Test Case 2
Description:
Precondition:
Flow of Event:
SETUP(A, B, Enterprise)
LRQ(A, B, Enterprise)
GRA(B, Enterprise)
Categories:
Settings Categories:
A
B
Enterprise
MasterRoutes.def
Enterprises.def
Network Topology
Interactions Categories:
SETUP(..,..,..)
LRQ(.., .., ...)
GRA(.., ...)
Post Condition:
Comment:

Test Case 3.1
Description:
Precondition:
Flow of Event:
[if (proper Enterprise)]
LCF()
SETUP(A,B, Enterprise)
LRA(B, Enterprise)
SETUP(A,B)
Categories:
Settings Categories:
A
B
Enterprise
MasterRoutes.def
Enterprises.def
Network Topology
Interactions Categories:
LCF()
SETUP(A,B, Enterprise)
LRA(B, Enterprise)
SETUP(A,B)
Post Condition:
Comment:

Test Case 3.2
Description:
Precondition:
Flow of Event:
[else]
LRJ()
RELEASE
Categories:
Settings Categories:
MasterRoutes.def
Enterprises.def
Network Topology
Interactions Categories:
LRJ()
RELEASE
Post Condition:
Comment:

Figure 20 Use Case Test Suite description

By interacting with the Cow_Suite tool, the test plan derived so far is then specified at the Test Procedures level, later in the design development, as described in Section 5.5.2. The designer can define the values of the choices and constraints for

all the Setting and Interaction Categories of the Test Cases, so that the Test Procedures, structured as in the Figure 21, can be automatically derived. In this case, a good understanding of the system and its details, characteristics and behaviours, is clearly required.

Finally, for each Test Case, a document called *Test Case Procedures Set* is drawn up delineating the set of the meaningful derived Test Procedures.

Test Procedure

SETUP(A,B, Enterprise)
call from a GON user to an external network when a correct association in MasterRoute.def exists
LRQ(A, B, Enterprise)
call from a GON user to an external network when a correct association in MasterRoute.def exists
GRA(B, Enterprise)
call from a GON user to an external network when a correct association in MasterRoute.def exists
A
e-mail
B
e164
Enterprise
Ericsson

MasterRoutes.def table
Network Topology files

Figure 21 One of the derived Test Procedures for Test Case 1

5.8.4 Comparison of Results

In this section we report the results of the comparison, with respect to both contents and development effort, between the two test plans focusing in particular on their peculiarities, strengths, and weaknesses It should be stressed that we do not evaluate the effectiveness of the two plans in terms of fault detection or time required for the real execution of test cases. When this experience was performed the test cases of ERI_TP had already been executed by ERI testers during the BRE pre-Integration testing and this phase closed. Hence we were not able to compare the ERI_TP and the UIT_TP on the basis of test results. We report only the pros and cons of the Cow_Suite in test generation, proving that it can be considered a valid instrument for defining test plans in the industrial environment.

However, due to the similarity of the tests (as detailed in the previous section) it has been possible to determine degree to which the requirements would be covered by the execution of the UIT_TP Test Procedures.

5.8.4.1 Comparison of the Contents of the Test Plans

In this section we compare the two test plans with regard to the following aspects: the degree of requirements coverage, and the expressiveness and the degree of detail of test cases derived for the same functionality.

Considering the requirements coverage, the two test plans achieve quite similar results, though some differences deserve note. The ERI_TP is surely the more accurate, because its test cases have been specifically constructed to cover all BRE requirements. Covering all requirements is assured by the thorough system knowledge of those who construct the test plan.

On the other hand, the degree to which UIT_TP covers the systems requirements is strictly linked to the SDs construction and information content. As described in Section 5.5.2, the UIT method can derive Test Cases only when well-formed, proper SDs have been furnished; the lack of SDs specifications prevents complete requirement coverage from being achieved.

Fortunately the case study analysed here had a quite complete UML system description; consequently, as seen in Table 6, UIT_TP provides almost the same requirement coverage as ERI_TP.

	TG1	TG2	TG3	TG4	TG5	UCTS1	UCTS2	UCTS3	UCTS4	UCTS5	UCTS6	UCTS7	UTCS8
RS_BRE1					X								
RS_BRE2	X	X				X	X				X	X	X
RS_BRE3	X	X				X					X	X	X
RS_BRE4	X					X	X						X
RS_BRE5	X					X							X
RS_BRE6	X					X							X
RS_BRE7			X					X					
RS_BRE8	X					X	X						X
RS_BRE9	X					X							X
RS_BRE10	X					X							X
RS_BRE11				X					X				
RS_BRE12					X					X			

Table 6 Requirement Coverage Matrix of ERI_TP and UIT_TP

Specifically in Table 6, considering the requirements of the BRE functionality, the columns labelled TG1.TG5 represent the different test cases groups in ERI_TP, while those labelled UCTS1…UCTS8 are the Test Cases set derived from the SDs associated with a specific UC (UCTS stands for Use Case Test Suite).

As shown by this table, the test cases both covering the RS_BRE1 or the RS_BRE3 requirement in the TG2 group (3rd row, 3rd column) of ERI_TP are not derivable via the UIT method due to the absence of the relative SDs. Actually these test cases were based mainly on the designer's experience and were built explicitly to test some exceptional conditions or anomalous system behaviours.

On the other hand, use of the Cow_Suite tool reveals some test cases not provided for in the ERI_TP (for instance RS_BRE3 requirement in the UCTS2 group, 4th row, 8th column). These Test Cases were derived from two SDs that describe the same objects' interaction from two different points of view. In this case, these Test Cases do not increase the requirements coverage of the UIT_TP, but represent a different way of testing the same functionality. One may choose between these equivalent Test Cases as necessary, for example by ease of implementation. Thus the two test plans show differences in expressiveness and degree of detail of the test cases

Table 7 shows a comparison of the two test plans in terms of derived test cases: the rows contain the ERI_TP test cases subdivided into groups, and in the columns the UIT Use Case Test Suites. An "X" in the cell signifies equivalence of the test cases based on the two methodologies. In particular, the UCTS7 and UCTS8 Test Cases are an alternative to the UIT derived Test Cases while test cases TG1-TC6 and TG5-TC2/3 are not provided for in UIT_TP.

Regarding the details of the two types of test cases, other differences can be noticed, in expressiveness and in the amount of information contained.

The ERI_TP test cases are clearly more thorough and detailed than those of UIT_TP. This is mainly due to the fact that it is an experienced designer who provides the ERI_TP descriptions. In writing such a document these experts draw on all their experience with and knowledge of the system components and interactions, and can therefore specify in detail the steps necessary for executing the test cases and providing a complete description of the environment and expected results.

On the other hand, as stated in Section 5.8.3.2, two different levels of detail can be distinguished in UIT_TP. The detail of UIT_TP at the Test Cases level is automatically derived using the information in the SDs; hence, the Test Cases contain only specifications of the operations without any reference to the environment or necessary preconditions.

ERI_TP - Test Group and Test Cases		UIT_TP - Use Case Test Suites							
		UCTS1	UCTS2	UCTS3	UCTS4	UCTS5	UCTS6	UCTS7	UTCS8
TG1	TC1	X							X
	TC2	X							X
	TC3	X							X
	TC4	X							X
	TC5		X						
	TC6								
TG2	TC1						X	X	
	TC2						X		
TG3	TC3			X					
TG4	TC1				X				
	TC2				X				
	TC3				X				
TG5	TC1					X			
	TC2								
	TC3								

Table 7 Comparison Matrix for Test Coverage

UIT_TP Test Cases are subsequently refined at the Test Procedures level by insertion of the Setting and Interaction values. In this way, the resulting UIT_TP is quite similar to the ERI_TP. However, the Test Cases' lack of complete description contained the fields labelled “Description”, “Precondition”, “Postcondition” are still apparent. The input of such information therefore requires specific intervention on the part of designers when they apply the UIT method. The UIT_TP specification at the level of Test Procedure is automatically derived with minimal interaction by the developers involved in the project. When the aforementioned fields are completed, the UIT_TP represents a detailed and complete reference document with the same expressiveness as the ERI_TP.

5.8.4.2 Comparison Relative to the test Plans Development

In this section we compare the time (considering an 8-hour working day) needed to formulate the two test plans and the effort required to transform the test cases into executable testing procedures. Moreover, we shall also consider the degree of system knowledge required to develop the two test plans and the software development stages in which they may be completed.

Considering ERI_TP, an evaluation of the time necessary to completely specify such documents is provided directly by the ERI Project Manager and the designers involved in the project. According to such assessments, 5 working days are necessary to complete an ERI_TP description, the work being divided into three separate parts:

- The first is test-case definition, which requires one day (8 hours) and involves only the designer. This phase consists mainly of analysing the system components in order to identify the possible test cases. The designer therefore constructs a testing schema for each interaction that should be tested.
- The second part is Procedures definition, which requires two days (16 hours). The designer must specify all the steps and actions necessary in order to check the system's interactions, particularly the description of the environment and definition of parameters.
- The ERI_TP definition ends with the refinement and completion of documentation. This two-day stage (16 hours) involves the designer and project manager, who must review the ERI_TP and correct any errors or inaccuracies.

The main advantage of the UIT_TP-based approach is that it is not necessary to spend time on formulating Test Cases; these are in fact derived automatically from UML design descriptions using the Cow_Suite tool. By simply executing the tool with the UML diagrams as input, the first part of the ERI_TP development cycle is completed immediately.

Completion of the UIT_TP, and therefore derivation of the Test Procedures, requires specification of the values of choices and constraints. We asked an ERI designer to work interactively with the Cow_Suite tool to insert the required information; this took two hours for data input. Therefore, deriving the executable Test procedures using the UML-based UIT methodology took two hours, as opposed to the 24 hours needed to complete the corresponding ERI_TP work.

At this point, the designer and Project Manager need only check the correctness of the derived Test Procedures and choose those to be actually run. The time necessary for these operations has been estimated at only one working day (8 hours). All told, derivation of the executable Test Procedure involves only 10 hours' time (one day and two hours) with the UIT methodology, while 40 hours (5 days) are needed for complete derivation of the ERI_TP. However, although the UIT_TP-derived test procedures can be passed on directly to the tester for the execution, they still lack the specifications regarding environments and pre- and post-conditions. If the Project Manager requires that such data be included in the UIT_TP, an additional day's work must be accounted for. In this case, the UIT_TP is derived in a total of 18 hours and the ERI_TP in 40 hours.

The two test plans also differ with regard to both the software development stages in which they can be completed and the knowledge required of the people involved

in test planning. As already stated, UIT_TP can be defined as soon as one or more SDs have been produced, i.e., during the analysis or design stage. In this case, any user with no particular system experience can automatically derive the Test Cases by simply applying the UIT method with the help of the Cow_Suite tool. It is worth noting that these Test Cases are not the final output of the UIT method application; they can be derived in an early stage of project development and therefore represent a tentative test plan, useful for the Project Manager for preliminary test scheduling and cost estimation.

As a matter of fact, Project Managers can construct a detailed preliminary test plan for pre-integration and other testing stages, both off-line and without designers' assistance. In particular, they can make decisions regarding the type of testing strategy to adopt, focusing on such strategies to fulfil requirements, provide code coverage or concentrate testing on the more peculiar functionalities. Moreover, by observing the types and structures of the Test Cases, Program Mangers can make a first prioritisation or even select some specific cases and thereby make an initial estimation about the number of Test Cases to be implemented during the actual testing phase.

During project development, when more detailed SDs and specific values for choices and constraint are available, the UIT_TP can be further refined via definition of Test Procedures. Specification of Test Procedures requires specific knowledge of the system generally available only to designers, who must specify the values for the Settings and Interactions Categories during the analysis or the design phase.

Regarding the ERI_TP and its description, two specific ERI staff members are involved: an expert designer, who is responsible for test-case derivation, and a Project Manager, who acts as supervisor and ultimate decision-maker with regard to acceptance of a Test Plan. The designer, on the other hand, must possess the necessary expertise to properly describe the test cases; therefore, an in-depth, thorough understanding of the system components and their behaviours is essential. Thus, the ERI_TP can only be drawn up at the end design phase, just before the testing phase, because a final, detailed description of all system components is required. In this way, only when the testing plan is completed can the Project Manager verify the degree of requirements coverage attained or the significance of the test cases derived and above all decide whether the test strategy adopted is suitable, or not. In the latter case, a different strategy must be adopted and consequently, a new testing plan prepared. From the point of view of the tester, the

two test plans differ from each other also in the degree of detail of the two types of test procedures. In fact, ERI_TP that requires the tester decide what the appropriate values are for each test procedure in order to attain requirements coverage, and therefore determine how many tests to run.

As a reference point, it is important to mention that the development and testing processes adopted in ERI are quite mature and well-established. ERI has been certified at CMM level 3 [PCC93]; therefore the test strategies that we compared to the Cow_Suite are effective and well-established.

5.8.5 Lesson Learned

This experience brought us to some interesting conclusions about the efficacy of the Cow_Suite Application. The main advantage was felt to be the fact that the Program Manager can exploit the provided UIT_TP as a baseline to adopt the most appropriate test selection strategy. UIT already provides the Project Manager with a detailed test plan already during the analysis or design phase, i.e., early with respect to the testing stage. Therefore, the Project Manager can get a realistic evaluation of the requirement and functional coverage that can be reached. If the values predicted are not satisfactory, corrective actions can be taken or a different choice of the proper test strategy for the testing phases can be considered. Moreover, the automated derivation of UIT_TP allows considerably reduction of the time necessary for test plan completion. In the proposed case study, we estimated a reduction of the time needed for the UIT_TP derivation of four times, while obtaining the same level of requirement coverage of the ERI_TP.

On the negative side, we observed that the automatic derivation of test cases failed to include the exceptional test cases, i.e., test cases to handle abnormal system behaviour. In particular, the UIT_TP missed two exceptional test cases, provided instead within the ERI_TP. Therefore, it would be opportune that before deployment, the UIT_TP is checked by an expert and additional test cases are possibly included to cover these special situations. This necessity is indeed common to any other automatic test case derivation method.

Concluding, the Cow_Suite tool has been quite favourably received within the ERI company which intents to apply the methodology in other test phases.

Summary

In this Chapter we presented an original approach, Cow_Suite implemented in a prototyped tool, useful for deriving and prioritising test cases starting from the UML specification. Cow_Suite integrates: a test strategy, Cowtest, which provides a practical help to managers for test planning both in case of fixed umber of test cases to be performed or fixed percentage of functionalities to be covered; a method, UIT_Sd, which constructs the Test Case and Test Procedures using solely the information retrieved from the Sequence and Collaboration diagrams available in the UML documentation.

Both for Cowtest and UIT_sd we detailed in the Chapter the requirements for their application, the procedural steps performed, the necessary user interactions, the typology of obtained results, and possible improvements. In particular we compared the Cow_Suite with the other similar approaches taken from the literature evidencing its main advantages: the use of exactly the same UML diagrams developed for analysis and design, the derivation of a test plan as early as possible in the development cycle, even during analysis or design phase), the definition of the Test Cases and Test Procedures in an incremental way refining them each time the degree of detail of UML diagrams considered increases, the capability to manage big test suites keeping under control their sizes and functional coverage.

We reported here our experience in the application of the Cow_Suite to two case studies, one taken from the literature and the other from a real industrial context. In particular the latter highlighted the usefulness of the Cow_Suite approach in test plan definition, giving in advance to the Project Manager a realistic evaluation of the requirement and functional coverage that can be reached during the testing phase in one-fourth of the time needed for the test plan derivation using of the conventional approach.

PART 4:
MEASUREMENTS FOR TESTING PHASE

6 Methodologies for Failure Prediction

Preface

In the previous Chapter we presented an automatic approach (Cow_Suite) which is useful, during the development process, for the testing phase organization. It is able to derive a Test Plan even from the UML documentation produced during the early stage of development, so that managers using Cow_Suite can schedule the cost and the effort required for the testing phase in advance.

Then, once the UML specification is completed and the testing phase effectively starts, the derived Test Plan can be completed and executed. This means running the derived Test Suites and collecting he test results, failures or successes.

At this point, it would be important to have methods for predicting during the execution of the tests, the final number of failure experienced up to the end of testing phase. Each failure requires meticulous extra work for finding and correcting the causing fault(s), which could lead to an enormous increase in the final cost of the testing phase. Knowing in advance the number of failures expected to occur up the end of this phase will permit swift corrective action, thus avoiding unpleasant surprises.

In this Chapter we focus our attention on the failure prediction, proposing new methodologies applicable in different situations. Specifically we concentrate only on non operational the test stage, i.e. excluding the results derived for instance from the beta or operational test (Chapter 2). We refer to Chapter 7 for more details concerning specific test phases.

6.1 Motivations

In spite of great advances in the software engineering field since complaints of a software crisis began to spread in the mid-seventies, the state of the art in software development is still such that producing defect-free code remains only wishful thinking. On the contrary, coping with software failures both during development

and after release, is one of the most difficult tasks of managers, while testing, debugging and maintenance activities still consume the major part of development effort and resources. For these reasons, methods for estimating the defects contained in software are of great interest for managers and testers.

Researchers have devoted much attention to this problem and have proposed many models for quantifying faults and failures, classified as “static” or “dynamic” approaches (see Chapter 2). Briefly, looking at properties of the present or past products, and/or at parameters of the development process, the former use these observations for estimating the total number of defects, or faults, in the current product. To this purpose a novel approach is presented in [CDM02], where the concepts and techniques from control theory are used for modelling the system test process and predicting its behaviour.

The latter observe defects, (or, more properly failures), as they show up in testing, and use statistical inference procedures to predict the number or the time of failures expected in future tests or in operation [BS96].

Thus, for prediction purposes, the static models are attractive to managers, because they provide "numbers", which the managers are eager for, very early in the process in comparison with dynamic models. These can only be used late in the life cycle, i.e., during the testing phases, when it may be too late to efficaciously re-direct development efforts. However, as mentioned in Chapter 2, the correct view is that static and dynamic models are both useful because they can be used in combination; i.e., the former in the front-end phases of the life cycle to allocate development time and resources, the latter in the final stages of development in order to evaluate the degrees of disturbance of the defects that inevitably remain, and to decide whether the product is ready for delivery[1]. To this purpose [CA98] proposed a model for predicting the remaining number of defects in the code based on the failures that are observed in testing, which is in a sense a hybrid approach between static and dynamic models.

Indeed, whether many or few, some defects will inevitably escape testing and debugging, so that, in the end, the only important measure for deciding whether a product can be released is software reliability [LY96]: i.e., the number of failures,

[1] To this purpose we refer to the Propean approach presented in the Chapter 4, which could be a valid aid in managing the testing phase organizations in terms of resources scheduling and personnel assignment.

and not of remaining defects, must be estimated. Unless they cause failures, remaining defects trouble neither customers nor producers.

However, in this Chapter we do not consider the reliability estimations for the motivations depicted below, but focus on dynamic models in order to evaluate the number of failures expected to be observed in future tests, based on the failures observed so far. In Chapter 7 the assessment of software reliability through testing is treated in detail.

Industrial test processes commonly undergo several subsequent steps (from unit to subsystem, and to system testing see Chapter 2) and eventually start operational testing only when the software configuration and behaviour are fairly stable. In particular industries have rarely applied the latter for testing single modules, or small subsystems, since identifying the required operational profile is quite difficult and expensive, and perhaps not sensible at all. They generally prefer to adopt the commonly used and less expensive test methods, e.g., branch coverage (Chapter 2), whose failure results do not comply with the underlying assumptions of the model for reliability predictions: i.e., if the test cases are randomly drawn from the operational profile, and as defects are found and removed, reliability will exhibit an increasing trend. However, even in the first stage of the operational testing both assumptions are hardly satisfied.

These are the underlying motivations for the work presented in this Chapter. In particular we develop some dynamic models that can be applied to predict the expected number of remaining failures in early test phases, without making assumptions as to how tests are selected.

The most attractive feature of these models is their simplicity: they merely require collecting the time intervals between subsequent failures. No estimation of parameters of the product or of the development process is needed, as will be described in Section 6.3. In particular, these models could be applied in combination with the Cow_Suite approach described in the previous Chapter, to evaluate the efficacy of the Test Suite execution in finding failures. However, here we present a more general description of these methods without referring to any particular strategy of test generation.

In the next section we present the basic idea used for the model definition: to predict the cumulative number of failures at the end of the testing phase by using the data collected during the testing phase itself.

6.2 Starting Point

In measurement, one tries to map observations of the empirical world to mathematical entities that can be formally manipulated. Models are defined attempting to capture one's intuition and understanding of the real world; indeed, "intuition is the starting point for all measurement" [FP97]. In this section we present the intuition underlying the dynamic models we have developed.

Originally the stimulus came from the analysis of the test results collected during several projects by a software producer, Ericsson Lab Italy in Rome (ERI). For each product this producer routinely logs the failures observed since early test phases until beta testing, and is interested in finding more effective ways to use these data for project management and product control. In particular the developer, who has a well-established and formalized test process, required that the model be compatible with its trouble-logging procedures, since it would have been difficult and expensive to modify them. It must be clarified beforehand that this producer was not looking for new testing methods to apply, that would facilitate failure predictions (as for instance would be the case if fault seeding approaches were applied). On the contrary, this producer wanted efficient metrics that could be applied to the data collected. It is plausible to assume that to a certain extent this proviso would be the same for many other producers.

So far, the data collected are used to derive measures of failure density, i.e., the ratio between the cumulative number of failures observed in a given time period and the product size, expressed in lines of code. Specifically, with regard to the results from beta testing, which is operational, standard approaches for reliability estimates and predictions can be applied, as described in detail in Chapter 7. In that Chapter a case study provided by the same producer is used for illustrating the application of software reliability engineering techniques.

As previously mentioned the models were developed considering the trouble-logging procedure used by this producer for data collecting, which registered the failure reports on a daily basis. Therefore we decided to group the failure data into test intervals (TIs), each one a daylong[2]. In particular a TI in which at least one failure is observed is called a *failed test interval (FTI),* otherwise it is called a *successful TI*. Note that no matter how small a test interval is chosen, until this

[2] The length of a *TI* depends strictly on the type of data collected and on the required granularity of the failure predictions. However if the failures are collected at intervals of length *l* a *TI* cannot be shorter that *l*.

remains larger than a single test, there will always be a chance to observe more than one failure within.

In simple words, the basic idea of the development of the models is that if *n* failures are detected after *t* TIs, this information can be used for estimating the cumulative number of failures at the end of test phase, assuming that we continue to test in the same way. In particular we suppose that the prediction may be different if the failures are uniformly distributed over the *t* TIs, or if instead all the failures are discovered in the first TI, and then the remaining (*t* - 1) TIs exhibit no failures.

Thus the time distribution of failure discoveries is hence a fundamental element for the models development. If in fact a priori knowledge or estimation of the failure detection rate is assumed over the sequence of TIs, say *fdr*, and *t* denotes the total number of TIs scheduled, obviously the expected number of failures *f* would be estimated by:

Eq. (1) $f = t \bullet fdr$

Of course this formula is rather naive and cannot be used in practice in this simplistic form, because the failure detection rate in testing can never be established with certainty; instead it is a random variable, for which a distribution should be identified. For each new product being tested, the empirical distribution of the failure detection rate can only be precisely drawn only after the testing is completed. However, if we could assume that, after having observed the test results for some time it stabilizes (i.e., it can be used as an approximation of the real, yet unknown distribution, to predict future behaviour), then a formula generalizing Eq. (1) could be used.

For deriving the cumulative number of failures at the end of test phase two different approaches have been implemented. The first, called the ***"One-Step Method",*** estimates the failure detection rate, using a statistical prediction method; that is to say, the *fdr* in Eq. (1) is treated as a random variable *D*, and uses an estimation of it to predict the expected number of failures *NF* (Section 6.4).

In the second approach, called the ***"Two-Steps Method"***, the expected number of failures *NF* is estimated in two subsequent steps: first we predict *NFTI*, i.e., the expected number of *FTIs*; second, from this estimation, we derive the expected number of failures *NF*. Correspondingly, at each step we introduce a random variable, for which an estimator has to be defined. This method was suggested from analysis of available data, and in particular from the observation of their variability (Section 6.5).

For prediction purposes, with for approaches (One-Step, Two-Steps), we have used a Classical estimator, i.e., relying on a frequentist interpretation of probabilities, and also an alternative, Bayesian estimator (Section 6.3.1), of a "subjective" interpretation of probabilities.

Finally the two methods were compared using a real case study and the results reported in Section 6.6. In particular we noticed that both the approaches perform better when the rate of detection of failures in testing remained more or less stable. This condition is clearly in contrast with the assumption underlying the models for reliability prediction. In this sense the approaches proposed in this Chapter are complementary to these and should be used when they cannot yet be applied.

Here, as previously indicated we want to provide methodologies for the early test phases, and in general to all those situations in which failures are found with some regularity, and remains valid only for limited periods. This means up to the point in which the rate of occurrence of failures starts to decrease, as a result of having removed a large number of faults.

It is worth noting that the predictions provided by the estimators are meaningless without a reference to control parameters over the development process. For instance, suppose that prediction brings to our attention an unexpectedly low predicted number of failures with respect to standard figures. This can be due either to an ineffective test process (bad news), or instead to a very good development process (good news). However in this Chapter we do not discuss the strategies that could be applied for discerning between these two situations since we focus mainly on prediction. However a possible solution is to incorporate the collection of useful invocation from similar projects, within the global strategy of project control and management, so that historical data can be used to set reference/target measures when necessary.

6.3 Background Knowledge

In this section we briefly present the fundamentals of the Bayesian theory (Section 6.3.1), used in the definition of the two methods (One-Step, Two-Steps), and the original Bemar model as presented in [Bm98] (Section 6.3.2), which was the starting point for the Two-Steps Bayesian approach. In Section 6.5.3 we show how the Bemar model has been included.

6.3.1 Bayesian Approach

Bayesian probabilities [GCS95] are attempted to describe an observer's subjective knowledge of yet-unknown events, and how this knowledge evolves as new events are observed. The Bayesian probabilistic ideas have been around since the 1700s. In 1713, Bernoulli recognized the distinction between two interpretations of probability:

- as the frequency of occurrence of an event in a sequence of repeated experiments, the commonly used interpretation in the frequentist (classical) theory;
- as a measure of the plausibility of an event about which knowledge is incomplete, the one that will be used in the Bayesian approach.

Therefore in the Bayesian framework an interval for an unknown quantity is directly related to the probability of containing the unknown quantity, while a frequentist interval can only be interpreted in relation to a sequence of similar experiments (but it does not give any estimation of the unknown quantity).

The basic idea of the Bayesian theory is due to Thomas Bayes, who in 1763 formulated the well-known Bayes' Theorem:

Eq. (2) $$P(X \mid Y) = \frac{P(Y \mid X) \cdot P(X)}{P(Y)}$$

Hence given an occurred event Y, and the conditional probability of event X on Y, denoted by $P(X|Y)$, the *posterior probability* is calculated using:

- the probability of event Y, denoted by $P(Y)$;
- the conditional probability of event Y given that event X occurred, i.e., $P(Y|X)$, called the *likelihood probability* , and
- the prior knowledge about the probability of event X, i.e., $P(X)$, called the *prior probability*.

To apply the Bayesian approach, first of all it is necessary to make explicit the prior belief into a prior probability distribution. In general, this is a difficult task, which also generates some perplexity about the usefulness of the Bayesian method [BS96]. The Bayesian method tells us how to treat observed data in order to derive the appropriate posterior distribution, but the identification of an appropriate prior distribution must come from outside. However this is the weakness as well as the strength of the Bayesian approach. The prior distribution expresses any desired prior state of knowledge, ranging between the two extremes of a virtual ignorance, called also *non-informative prior* or *least informative priors*, and highly informative

knowledge or *conjugate priors.* Conjugate families are those distributions for which the prior and the posterior distribution are members of the same family[3]. In this case, computing the posterior probability is usually trivial. We report in Section 6.3.1.1 the Gamma Poisson Model which is one of the conjugate families used in the rest of this Chapter.

Generally the choices for deriving a prior distribution can be summarized as the following possibilities:

1. if evidences or data are not available, the prior probability can give equal probability to each possible value (for example by using a uniform distribution). In this case, the prior probability is a non-informative prior;
2. if a certain amount of data has been previously collected, it is sometime possible to derive the prior probability directly from them (for example by associating a suitable normal distribution);
3. if there is a strong belief in certain homogeneity in the way in which the priors probabilities change as more evidence is acquired, a family of conjugate prior can be used for the posterior probability computation.

In contrast, the Classical approach has a broad field of application and can be used whatever is the behaviour of the product being tested. But this is also a limitation of the method, because it does not allow us to exploit the evidences of collected historical data collected which could contribute to more accurate predictions.

In this Chapter we use both the Classical and the Bayesian approach in the models definition. In particular we compare these models on the prediction of the cumulative number of failures at the end of testing phase. The purpose is to verify if the Bayesian estimators could perform better than the Classical because they need fewer data to obtain valid predictions, and exploit the available knowledge about the rate of occurrence of failures.

6.3.1.1 The Gamma Poisson Model

Sometimes it is possible to assume that the failure will occur "purely randomly", i.e. in a simple Poisson Process of rate λ. In this case the number of failures, *R*, at time *t*, has a Poisson distribution [LO87]:

$$P(R = r) = \frac{(lt)^r e^{-lt}}{r!}$$

[3] Formally if *F* denotes the family of probability density functions (pdfs) f(x|θ), a family Π of prior distributions for θ is called a ***conjugate family*** for Φ if the posterior distribution for θ is in the class Π

and in particular $P(R=0)=e^{-lt}$.

The conjugate family here is the Gamma. Thus if we represent *a priori* belief about the failure rate λ by *Gamma(a,b)* with expected value *a/b*, the posterior for λ after seeing *r* failures during time *t* is *Gamma(a+r,b+t)*:

$$f(l \mid r,t,a,b)\frac{(b+t)^{a+r}\, l^{a+r-1}e^{-(b+t)l}}{\Gamma(a+r)}$$

with expected posterior value *(a+r)/(b+t)*, where Γ(a+r) is the gamma function defined as

$$\Gamma(a+r)=\int_0^{\infty} x^{a+r-1}e^{-x}dx$$

6.3.2 Bemar Method

The Bemar model, presented for the first time in [BM98] derived the cumulative number of failures at the end of the testing phase in two steps: first the number of failed test intervals, N_{FTI}, is derived and then, from this number, the number of failures N_F is calculated.

Considering the testing phase established to be *NTI* test intervals long, a random variable *T*, taking discrete values within the interval [*1*, *M*] (where *M* is the maximum value), denotes the distance (in terms of *TI*s) between two subsequent FTIs. Precisely, for each *i* within [*1,M*], the associated probability mass function [LO87], (*pmf*), $p_T(i) = P(T=i)$, gives the probability that the next failure will be observed after i^{th} *TI* is a *FTI*). Denoting the expectation of the r.v. *T* by

Eq. (3) $$E[T]=\sum_{i=1}^{M} P_T(i)$$

the following formula holds[4]:

Eq. (4) $$NTI = N_{FTI}\cdot E[T]$$

and solving it for N_{FTI}, we obtain:

---

for all f ∈ Φ, all priors in Π, and all x.

[4] Actually, this formula holds precisely if it can be assumed that the last test interval is a failed one. Otherwise, the left-hand side should be decreased by the number of test intervals occurring between the last FTI and the last test interval.

Eq. (5) $$N_{FTI} = \frac{NTI}{\mathrm{E[T]}}$$

It is worth noting that this formula requires a procedure to derive *E[T]* for instance based on data collected from similar products. In the Bemar model a Bayesian approach is used for this purpose. As mentioned in the previous section, in the Bayesian framework, probabilities are meant to describe an observer subjective knowledge of yet-unknown events. This knowledge evolves as events are observed. In this context, the *pmf* of *T* $p_T(i)$ is taken as the *prior* knowledge about the behaviour of a product under test, i.e., $p_T(i)$ is taken to model a tester's subjective belief about the rate of failure detection *before* some evidence (the test results) about the product under test is observed.

In particular the realization of a sequence of test intervals with and without failures during the testing phase is observed. Thanks to this evidence, the tester's knowledge about this product evolves and the posterior distribution for the *pmf* of *T* can be derived. Denoting by F_K the sequence of observed outcomes (failed/successful) for the first *k* TIs, the posterior distribution $p'_{T,k}(i)$, is derived as $P(T=i \mid F_k)$, i.e., it is the update of $p_T(i)$ after having observed the sequence F_K. Applying Bayes' formula we obtain:

Eq. (6) $$p'_{T,k}(i) = (P(T = i \mid F_k)) = \frac{P_{prior}(T = i)P(F_k \mid T = i)}{\sum_{j=1}^{M} P(F_k \mid T = j)P_{prior}(T = j)}$$

The term $P(F_k|T=i)$ corresponds to the likelihood function and can be derived considering, if $T=i$, then the probability of observing a failure in the next test interval is $1/i$, i.e.:

Eq. (7) $$P(F_1 \mid T = i) = \begin{cases} \frac{1}{i} & \text{if } F_1 \text{ is failed} \\ (1 - \frac{1}{i}) & \text{if } F_1 \text{ is successful} \end{cases}$$

Substituting this in Eq. (6), and iterating the same reasoning also to the subsequent test intervals, we finally obtain[5]:

Eq. (8)
$$p'_{T,k}(i) = \frac{p_T(i)\cdot\left(\frac{1}{i}\right)^f\left(1-\frac{1}{i}\right)^{k-f}}{\sum_{j=1}^{M} P_T(j)\cdot\left(\frac{1}{j}\right)^f\left(1-\frac{1}{j}\right)^{k-f}}$$

which gives the posterior *pmf* for the random variable T, after observing *k* test intervals, out of which *f* were failed. This formula is used for deriving the *E[T]*.

By using the Eq. (5) it is possible to then derive $N_{FTI,k}$, i.e., the number of *FTI*s expected after *NTI* test intervals, using the test information collected during the first *k* test intervals.

From $N_{FTI,k}$ the total number of failures N_F now needs to be estimated. This clearly depends on how many failures on average are observed within a *FTI*. We can again define a random variable *F* to represent the number of failures observed within a *FTI*, and then derive N_F from N_{FTI}, with $N_F = N_{FTI} \bullet E[F]$

The empirical *pmf* for *F* can be derived by considering the results from the first *k TIs* as well as the maximum number of failures per *FTI*, called *MF*. From the distribution of the number of failures within a failed test interval, we are able to calculate the expectation:

Eq. (9)
$$E_k[F] = \sum_{i=1}^{MF} P_k(F = i)\cdot i$$

Therefore, after having observed *k TI*s, the number of failures that a product will show at the end of the functional test is:

Eq. (10)
$$N_{F,k} = N_{FTI,k}\cdot E_k[F]$$

The Eq. (5) and Eq. (10) are to be used incrementally during the testing phase, i.e., considering each time a greater value for *k*, and adjusting the *pmfs* involved correspondingly. In this way, the prediction about the total number of failures for a product as testing proceeds will increasingly precise.

[5]In the generalization of this formula from the case *k*=1 to larger values of *k*, we have in reality used some relaxed assumptions, which could raise some objection to its validity from a purely theoretical viewpoint. However, applying this formula to different cases study we notice that it performed better than other theoretically stronger models.

6.4 One-Step Method

In this section we present the One-Step method [LPM99, BMM02a] as described in the main steps in Figure 1. This figure is valid for the one-step method based both on a Classical approach (Section 6.4.1) and on a Bayesian approach (Section 6.4.2).

As indicated in Section 6.2 we estimate the number of failures over a fixed test period, specifically after *NTI* test intervals. Therefore the first steps in both models are collecting data and then defining a random variable *D* to denote the daily failure detection rate. Using a valid estimate D_e of *D*, we obtain the estimated number of failures, over a *NTI* long period of test intervals, as $NF = D_e \bullet NTI$

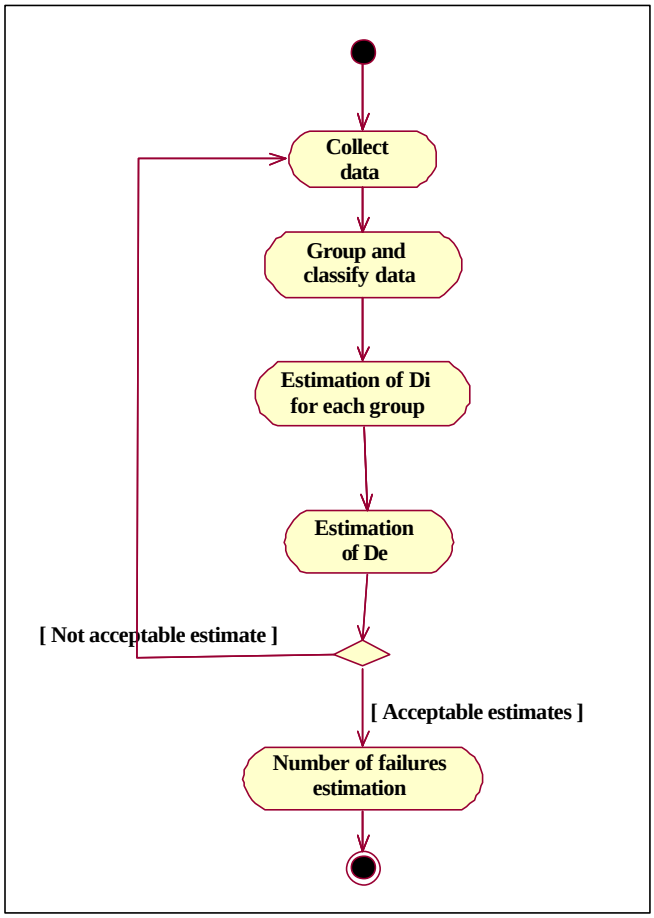


Figure 1 Main steps for the One-Step method

6.4.1 One-Step Classical

Referring to Figure 1, after data collection we decide to group the test intervals into separate sets containing the same number *t* of *TI*s . Different possibilities can be

evaluated depending on the typology of the data collected and on the desirable granularity of the final predictions. For instance if the *TI* corresponds to an hour a possibility is to group the test intervals into a daily base (considering a working day of 8 hours long we set t =8 and therefore we put 8 *TIs* in each group). Otherwise, if a *TI* is equal to a working day, as our original data, we can divide the *TIs* in groups of 5 test intervals, corresponding to one calendar week of testing. In the former case we will upgrade the predictions of the final number of failures at the end of the testing phase daily, in the latter weekly.

We assign then to each group of test intervals an increasing identification number k and for each of them we derive the number of failures experienced within the *t TI*s. To this purpose for each group we define an estimator D_i of the failure detection rate of the TIs in that particular group (e.g. per hour failure detection rate if the *TI*s correspond to one hour, daily failure detection rate if *TIs* are equal to a day), which is calculated as D_i = (number of experienced failures within the group i)/ t. To derive the estimator D_e after having observed $k \cdot t$ *TIs* we use the sample mean of the values D_1, D_2, ...,D_k, i.e., $D_e(k) = E_k[D]$, which is as an unbiased, consistent, and the minimum variance unbiased estimator of D [AL90]. The estimate obtained is evaluated by the *confidence interval*, that is a probability judgment about the accuracy of the estimate delivered.

We continue repeating the steps described above, i.e., we wait for the data relative to another group of *t TIs*, increment k and repeat the above procedure, until the desired level of accuracy is reached.

Assuming this happens after a certain number k^* of groups of *t TI*s, we can use the estimated failure detection rate $D_e(k^*)$ to make the predictions, i.e., after *NTI* test intervals, the global number of failures expected at the end of testing phase can be obtained by $N_{F,k^*} = D_e(k^*) \cdot NTI$.

The main limitation of this approach lies in the fact that a large amount of data is necessary in order to derive sensible confidence intervals. Oftentimes the value k^* of groups of *TI*s guaranteeing the desired level of accuracy can be so high that it makes this kind of approach impractical.

6.4.2 One-Step Bayesian

In the One-Step Classical approach, we assemble the test intervals in groups of t and assign to each group an increasing identification number k. In this case after

observing the *k-th* (current) group of failure data, we compute the cumulative (i.e., from group 1 to group *k* inclusive) number of failures.

To derive an estimation D_e of the failure detection rate, we use this value as a parameter of a suitable statistical model (Gamma/Poisson see Section 6.3.1.1), by which we calculate the expected cumulative *NF* over a future period of testing. In our specific context the Gamma/Poisson model results the most suitable statistical method, but depending on the data available, it is possible either to use other conjugate families of distribution, or apply one of the methods described in Section 6.3.1. Using the Gamma/Poisson model, as described in Section 6.3.1.1, if *Gamma(a,b)* represents the prior belief (for some suitable choice of the parameters *a* and *b*) the posterior belief about the failure rate *D* is represented by *Gamma(a+x, b+t)*, where *x* is in this case the number of failures observed in a time interval *t.*

We then use the relative error between two subsequent estimates to evaluate the accuracy of the estimate obtained. Until the desired level of accuracy is not reached, we wait for the data relative to another group of *t TIs*, increment *k* and repeat the steps described above.

We then use the outcomes collected during the first n^*, i.e. $k \cdot t$, test intervals to derive $E_{n*}[D]$, i.e. the posterior expectation of *D* after n^* *TI*s. This is taken as the estimator $D_e(n^*)$ to derive $N_{F,n*}$, i.e., the predicted number of failures after *NTI* test intervals.

6.5 Two-Steps Method

The Two-Steps method predicts the number of failures at the end of testing phase in two steps by applying the statistical control procedure described below [LPM99, BMM02a]. In particular we first predict *NFTI*, i.e., the expected number of *FTIs*; then from this estimation, we derive the expected number of failures *NF*. Correspondingly, at each step we introduce a random variable, for which an estimator must be defined. Figure 2 shows the main steps of the applied statistical procedure, which will be described in detail respectively in Section 6.5.2 for the method based on a Classical approach, and in Section 6.5.3 for that based on a Bayesian approach.

6.5.1 Prediction Procedure

We use a random variable Q to denote the probability that the next TI will be a failed one and derive a valid estimate Q_e of Q. Over a NTI long period of test intervals we then obtain the number of failed test intervals as: $NFTI = NTI \cdot Q_e$.

Once a value for $NFTI$ is thus estimated, the total number of failures will clearly depend on how many failures on average are observed within a FTI. Therefore we introduce another random variable F to represent the number of failures observed within a FTI, and then using a valid estimate F_e of F we derive NF from $NFTI$ simply as $NF=NFTI \cdot\cdot F_e$.

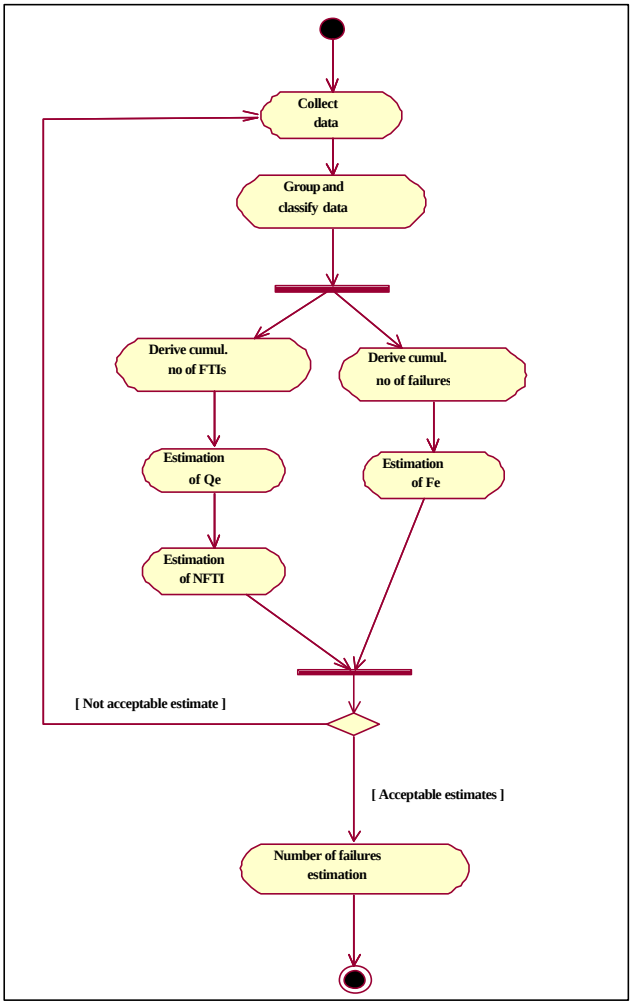


Figure 2 Main step for the Two-Steps method

As concerns the estimation of F_e, one possibility is to adopt the classical estimator $E[F]$, based on the sample mean (also called arithmetic mean) of the

observed failures over the number of observed *FTI*s, which for large samples, has the property of consistency and unbiasedness [AL90]. Of course other possibilities can be launched, depending on the typology of the data available. In this section we decided to adopt the sample mean because when we applied the Two-Steps method to the available case study, it was the one that first stabilized.

As for the One-Step Bayesian method, we consider the test intervals assembled in groups of t and assign to each group an increasing identification number k. After observing the k^{th} (current) group of failure data we derive two different values: the cumulative (i.e., from group 1 to group k inclusive) number of *FTI*s and the cumulative number of failures. These values are used as parameters of different statistical models to derive: an estimation Q_e of the probability that the next test interval will be a failed one; an evaluation F_e of the number of failures in each test interval; the predictions of global *NFTI* and global *NF* over a future testing period.

We use classical statistical techniques (e.g., confidence interval, relative error) to evaluate the accuracy of the estimates obtained. As above, until the desired level of accuracy is not reached, we wait for the data relative to another group of t *TI*s, increment k and repeat the procedure described.

6.5.2 Two-Steps Classical

To derive the probability Q that the next *TI* is failed, given a sample of *NTI*, we based the Two-Steps Classical on the maximum likelihood estimate [AL90, LO87]. The idea underlying the *maximum likelihood* estimate of a parameter is to choose that parameter value which makes the observed sample values the most probable.

In this case the sample to be analysed is formed by sets of test intervals of size n (with n=5, 10, …, *NTI*) with or without failures. We can depict the sample as a sequence of Bernoulli trials [LO87] with probability Q of failure on each trial (note that in such a way we are assuming independent *TIs*, which is reasonable for the approach followed in test selection).

Thus, if the observed number of failed *TIs* is f, then the maximum likelihood estimate of Q is given by [AL90, LO87]:

Eq. (11) $l(Q) = Q^f (1 - Q)^{n-f}$

The maximum likelihood estimate of Q is that value of Q which maximizes the likelihood function l, or its logarithmic form. Solving for Q yields the maximum likelihood estimate results:

Eq. (12) $Q_e(n) = f / n$

It can be proved that such is an unbiased, consistent, and the minimum variance unbiased estimator of *Q* [AL90].

To complete the statistical control procedure, we associate to each Q_e its confidence interval. As in the One-Step Classical method, the study of the confidence intervals leads us to determine that, after a certain number n^* of *TIs*, the desired level of accuracy is reached. We in fact are dealing with a random variable, so we cannot predict with certainty that the true value of the parameter, *Q*, is within any finite interval. However we can construct a confidence interval, such that there is a specified confidence or probability that the true value *Q* lies within that interval. Of course for a given confidence level the shorter the interval, the more accurate the estimate.

It can be proved [AL90]. that, for a sample of large size *n*, an approximate *100(1-a)%* confidence interval for the Bernoulli parameter *Q* is given by:

Eq. (13)
$$Q_e(n) - z_{a/2}\sqrt{\frac{Q_e(n)(1-Q_e(n))}{n}} < Q < Q_e(n) + z_{a/2}\sqrt{\frac{Q_e(n)(1-Q_e(n))}{n}}$$

where $Q_e(n)$ is calculated as described previously and values for the parameter $z_{a/2}$ are found in statistical reference tables [LO87]. Therefore once a confidence level is fixed (90%) according to the producer exigencies, it is possible to associate to each $Q_e(n)$ the relative confidence interval estimated after *n TIs*. The study of the confidence intervals leads us to determine that, after a certain number n^* of *TI*s, the desired level of accuracy is reached. Therefore we can use the estimate $Q_e(n^*)$, obtained after n^* *TIs* , to make predictions After *NTI* test intervals, the number of failed test intervals *NFTI* is obtained as $NFTI_{n*} = NTI \cdot Q_e(n^*)$.

As a final step we obtain the total number of failures expected at the end of the testing phase as $NF_{n*}=NFTI_{n*} \cdot F_{e,n*}$.

Again the main limitation of this approach lies in the fact that a large amount of data is necessary to derive significant confidence intervals. Consequently the value n^* of *TIs* guaranteeing the desired level of accuracy can be quite high.

6.5.3 Two-Steps Bayesian

The Two-Steps Bayesian is basically the application of the Bemar model described in Section 6.3.2. In this case the random variable *Q* denoting the probability that the next *TI* is failed, corresponds to the variable *T* mentioned in the Bemar model. Here we only report the formula used for the posterior distribution of

the random variable *Q*, in which we denote with F_n, the sequence of observed outcomes (failed/successful) for the first *n TIs*, and with *f* the number of *FTIs* observed in the sequence F_n,

Eq. (14)

$$p'_{Q,n}\left(\frac{1}{i}\right)=\frac{p_{prior(Q)}\left(\frac{1}{i}\right)\cdot\left(\frac{1}{i}\right)^{f}\left(1-\frac{1}{i}\right)^{n-f}}{\sum_{j=1}^{M}P_{prior(Q)}\left(\frac{1}{i}\right)\cdot\left(\frac{1}{j}\right)^{f}\left(1-\frac{1}{j}\right)^{n-f}}$$

As described in the prediction procedure (Section 6.5.1), we use this updated distribution to derive, $E_n[Q]$, i.e. the posterior expectation of *Q* after *n* observed *TIs*. This is taken as the estimate of $Q_e(n)$ to derive $NFTI_n$, i.e., the predicted number of *FTIs* expected after *NTI* test intervals, based on the test outcomes collected during the first n test intervals, and on the prior expectation about *Q*. From $NFTI_n$ the expected number of failures can then be derived in the same way as in the Two-Steps Classical method.

6.6 Application Results

The One-Step and the Two-Steps methodologies can be applied without referring to any particular test strategy. The failure data therefore could be obtained from the execution of the Test Suites derived by the Cow_Suite tool (see Chapter 5) as well as other specific methodologies. In particular, here we report the application of the One-Step and the Two-Steps approaches to a case study provided by the Ericsson Lab Italy in Rome (ERI) and described in the next section. The purpose in this case was to increase the ERI capabilities in statistical process control and in prediction methods. We describe the results obtained from this case study in Section 6.6.2.

In the preface of this Chapter we specified that the methodologies presented here were specifically built for dealing with non-operational data. However, their general nature also led us in the application of the Two-Steps Bayesian methodology to some operational results provided by ERI. We report the description of this experiment in Section 6.6.3. Naturally this type of data is more suitable for the application of the models for the reliability predictions, as will be discussed in detail in Chapter 7.

6.6.1 Case Study

In agreement with the ERI producer, we first selected a set of strategic processes to be used for the application of the One-Step and the Two-Steps methods [LPM99,

BMM02a]. Of course, applying statistical process control techniques to all development processes was not economically feasible. Therefore, the Function Test, which is one of the four test phases in ERI test strategy (namely Basic Test, Integration Test, Function Test and System Test (Chapter2)), has been identified as a strategic one in order to meet commitments to customers with respect to quality objectives. In particular, the Function Test is the phase in which the system functions are verified.

One ERI objective is to reduce by a determined amount the failure density in operation, which is obtained by monitoring the first six months of operation of released products. Failure density is measured by the ratio between the cumulative number of failures observed by the client in those six months and the product size, expressed in lines of code.

Root Cause Analysis (RCA) of reported failures is routinely performed, to track back failures to their causes, the faults, and to the phase in which the latter originated. An important finding of RCA for ERI products was that a high percentage of failures (48%) corresponded to software faults that could have been discovered during the Function Test phase. Therefore one of the actions to reduce failure density is to decrease the number of failures "slipping through", i.e. the ratio between the number of failures found during first six months, and the sum of failures found during Function Test and the first six months.

Currently, Function Test is performed accordingly to a function test specification, with the goal of testing conformance of the target function to its specification. Testers derive test cases manually, by making a systematic analysis of the specification documentation and attempting to cover all the specified functionalities (or use cases). This means that the test cases are deterministically chosen by examining the functional specifications before test execution starts (which also implies that the number of tests to be executed is decided in advance).

Function Test execution is organised in a specified number of stages. The tests are executed during working days (i.e., five days a week) and 8 hours per day. All the failures discovered within a test stage are logged and reported to software designers, who trace failures back to the code and correct them. A new software version is then released, which is resubmitted to testing in the next test stage. For each project, the information registered consists of the Start and End dates of the test phase, and of the calendar day (but not the time of day) of discovery of each failure.

Therefore, for our purpose we collect the failure data of several projects subjected to the Function Test process and use them to apply the One-Step and Two-Steps prediction method. The size of the software under test varies from project to project (minimum 50 kloc, maximum 150 kloc), and the failures in the data sets considered were classified as priority B (major failures).

For each project the failure data observed in the first part of the test process, grouped into test intervals (TIs) each one a day long, are used to predict the expected cumulative number of failures over the planned period of Function Test. This in particular stops when all the test cases defined in the test case specification have been successfully performed, either at the first try or after fault repair. Specific exit criteria related to the measured rate of failures detected over the testing period were not explicitly considered in the test process before this experience, and no estimation of the remaining number of faults was performed. It is worth noting that a very important property of prediction systems is the speed of convergence of estimates. With respect to this, we compare the performances of the estimates of the One-Step and Two-Steps methods (Sections 6.6.2 and 6.6.3).

Parameters Setting

Some historical data, derived from similar products subjected to the same Function Test process have been used to set the various parameters required by the models considered. Specifically considering the One-Step Bayesian, from an accurate analysis of the failure behaviour of several products we observed that we could group the products in classes depending on the average failure rate D they exhibited at the end of the test phase. We could thus derive the proper parameters (a, b) of the prior Gamma distribution for each group.

In the Two-Steps Bayesian, when analysing the failure data we noticed that the distance between subsequent FTIs was not greater than 20; therefore we considered that the variable Q could take discrete values within the interval [1,20]. In particular a proper prior distribution was obtained observing that for all products considered the distribution of Q concentrated for most of its realizations on the three same consecutive values, while very rarely took the other possible values.

6.6.2 Result Analysis

In this section we provide a few examples of the results obtained from the use of One-Step and Two-Steps methods as described in the Section 6.5. We do not report in the following figures the results of the application of the One-Step Classical

method, because in this particular context this approach required too many data for producing acceptable estimates.

In the following diagrams, on the horizontal axis we put the number of elapsed groups of TIs. On the vertical axis we put the cumulative number of failures over completion of the scheduled test period (for confidentiality reasons, we omit the actual numbers). When a prediction becomes acceptable based on the confidence interval or the relative error, as described in the prediction procedure, we stick to it and the prediction curve becomes a straight line. We check a posterior the prediction of the models against the actual number of failures observed at the end of the test period (the dotted horizontal line) (of course this knowledge is in no way used to make the prediction). The strip marked with vertical segments around the latter indicates the zone where the relative error of the prediction would be below 10%.

In Figure 3 we show the results for Project 1. In particular the curve labelled "Gamma/Poisson Model" corresponds to the result obtained for the One-Step Bayesian method, the one labelled "Our Bayesian Model" to the Two-Steps Bayesian, and the one labelled "MaxLik (Classical)" to the Two-Steps Classical.

Analysing the curves obtained it is possible to observe that the maximum likelihood method produces a valid prediction after 14 groups of TIs (70 testing days); the Two-Steps Bayesian model and the Gamma/Poisson model anticipate the prediction, respectively of as many as four and two groups. That is a very good result from the manager's point of view. With regard to the outcome of prediction, both models produce valid estimates (for us, this meant within the 10% error strip).

In a second project considered, as shown in Figure 4, we can see, that both the estimates produced with the Gamma/Poisson and the Classical model models are outside the 10% error strip. In general we could see that the effectiveness of the Gamma/Poisson model is greatly dependent on the choice of the parameters (a,b).

We must add, for thoroughness' sake, that unfortunately the Bayesian models (One-Step and Two-Steps) did not consistently work better and faster for all data sets. In a few circumstances the Classical model performed better. Finally, sometimes it was quite difficult to either find a suitable prior distribution or proper parameters (a,b) of the Gamma distribution.

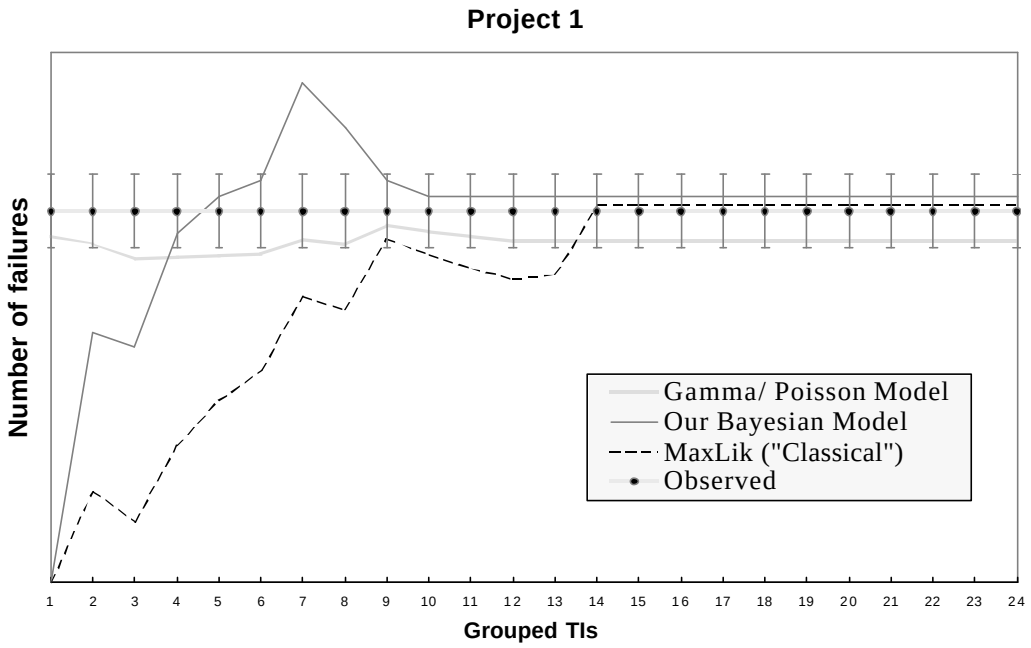


Figure 3 Prediction results for Project 1

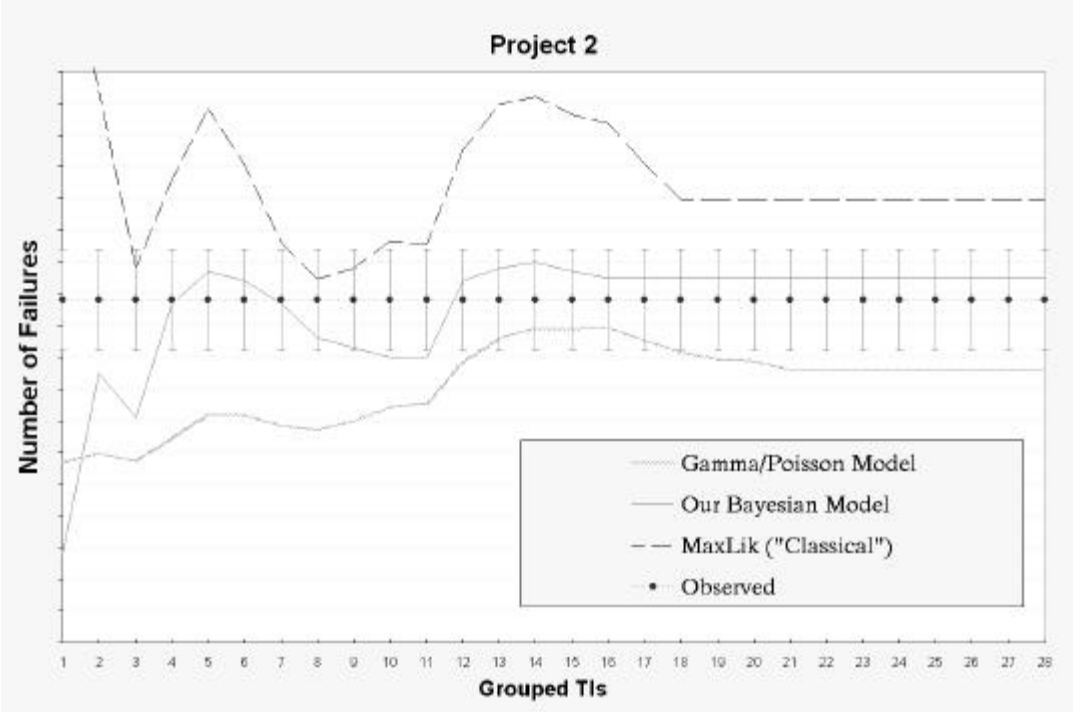


Figure 4 Prediction results for Project 2

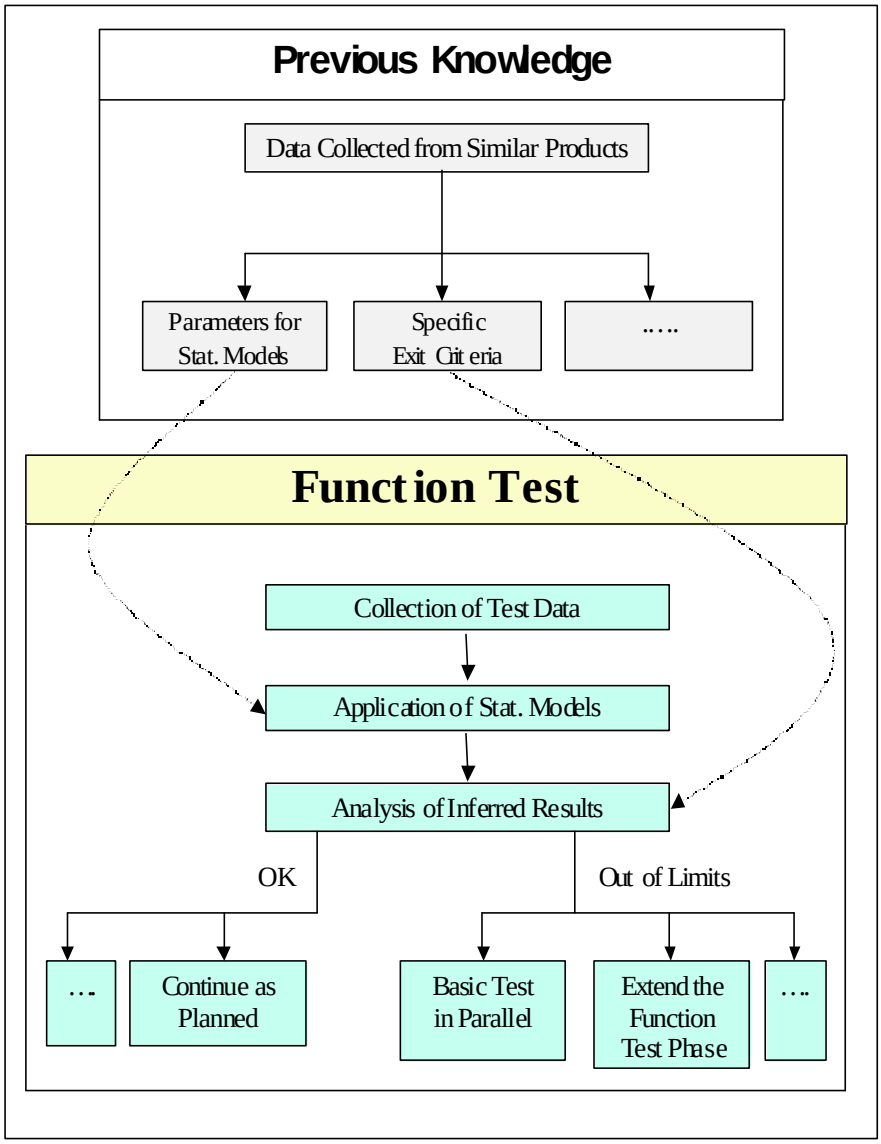


Figure 5 Main steps of the statistical process control strategy

So we took the approach of applying all the estimators in parallel (this can be done cheaply by a tool) and choosing the one that first reaches a valid prediction. From our perspective, such models provide the project management team with an effective and inexpensive means to take corrective actions when causes of variation are identified with respect to the Function Test process performance baselines (e.g., minimum and maximum failure density computed on historical data in the same product line), and with respect to meeting ERI "slipping through" objectives.

As for the other methods proposed in this Thesis, (Propean in Chapter 4, Cow_Suite in Chapter 7) also in this case in the definition of the One-Step, Two-

Steps models does not require any unnecessary additional work from the people who are going to use the proposed strategies, e.g. testers. On the contrary we try to conform as much as possible the models to the existing data collection procedures.

Our philosophy is always to propose methodologies that are easy to use and readapt and that require any, no or at least low cost and effort to be applied in the real industrial context.

The implementation of appropriate corrective actions (such as executing an extended Basic Test in parallel to the Function Test, or postponing the end date of the Function Test) can mitigate the risk of failures slipping through the Function Test during the first six months in operation, thus reducing reworking and maintenance costs. In Figure 5, we illustrate the main steps in the application of the statistical process control techniques discussed here.

6.6.3 Two-Steps Bayesian Model with Operational Data

As mentioned in the introduction of this Chapter, the Two-Steps Bayesian model has also been applied to some operational test results collected by the same producer during beta testing (for which we did no expect the model to work as well as for functional testing). We selected the Two-Steps Bayesian because it was the model that performed better in the majority of the cases during the functional test phase of several products. The intent was discovering if and how the Two-Steps Bayesian model could be applied either, as a complementary approach to reliability growth models, or in those situations in which the failure data relative to operational testing did not show a reliability increasing trend (see Chapter 7).

One of the major difficulties in applying the Two-Steps Bayesian model to this kind of data was the definition of the prior probabilities. In fact we were deficient in operational test results collected previously on similar projects. In this situation we could not apply the criteria described in the previous section for the selection of a prior pmf of T; we thus decided to adopt a uniform prior distribution.

For the rest, the approach of applying the Two-Steps Bayesian model to the data collected during the operational phase is the same as that described in Section 6.5.2. Also in this case the failure data was collected on a daily basis, therefore we again grouped the test intervals into set of 5 *TIs* corresponding to a week of testing. We report the results in the figure below where in this case k indicates the different *TIs*.

As shown by this figure the performance of the model becomes acceptable after 110 *TIs*, over an overall e period of 180 *TIs*. In this case we do not stick to this value

as in the previous examples, because we were also interested in the general behaviour of the Two-Steps method as the testing proceeded. We want to analyse the improvement of the prediction as more data failure data become available. As shown in Figure 6, after the 125 *TIs* the prediction stabilized between the 5% error strips. This represents a valid result from the manager's point of view, and we would expect better results using an informative prior probability.

To compare our results with those provided by the standard models for reliability prediction, we attempt to use the latter with the same data set. Unfortunately we were not successful; the problem was that the reliability did not regularly increase, as required by those models (see Chapter 7). Thus we expect a worse performance of the Two-Steps Bayesian model with data that exhibit consistent reliability growth.

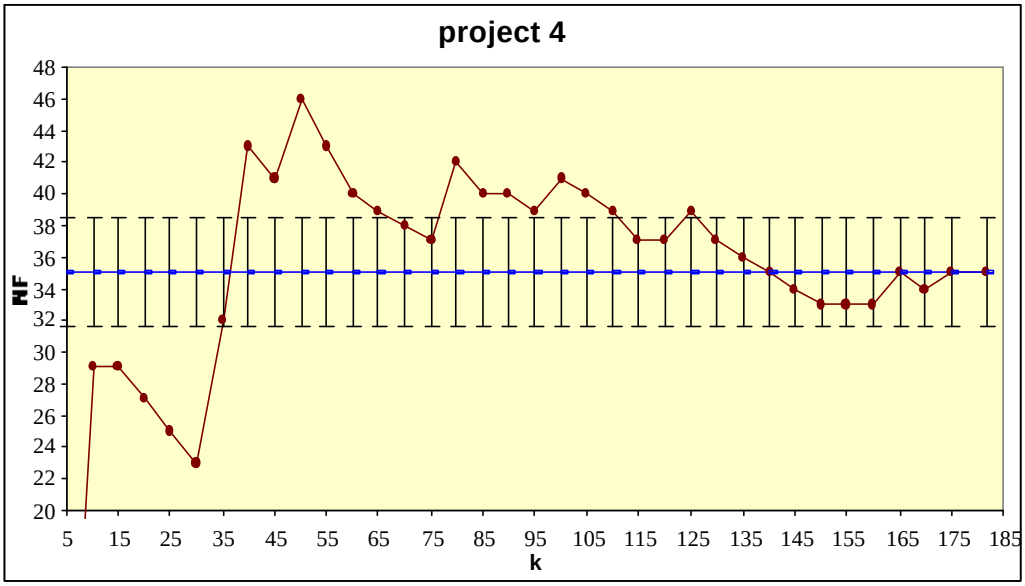


Figure 6 Prediction of the Two-Steps Bayesian model for beta testing

To verify this hypothesis we have tried the model on a set of data taken from the literature [ACL96]. These data are reported as execution times in seconds between successive failures; therefore, in order to apply our model, we grouped the failure data into test intervals of 600 seconds. Also, in this case we used a uniform prior distribution because we had further information about the failure behaviour.

The results are shown in Figure 7 where *k* indicates the corresponding group of *TIs*. As expected, the results became acceptable only late in the operational phase, therefore the predictions provided by the Two-Steps Bayesian model are not very useful.

From this experience, we can conclude that the Two-Steps Bayesian model could represent a complementary approach to reliability growth models when the hypothesis of applicability of the latter are not verified.

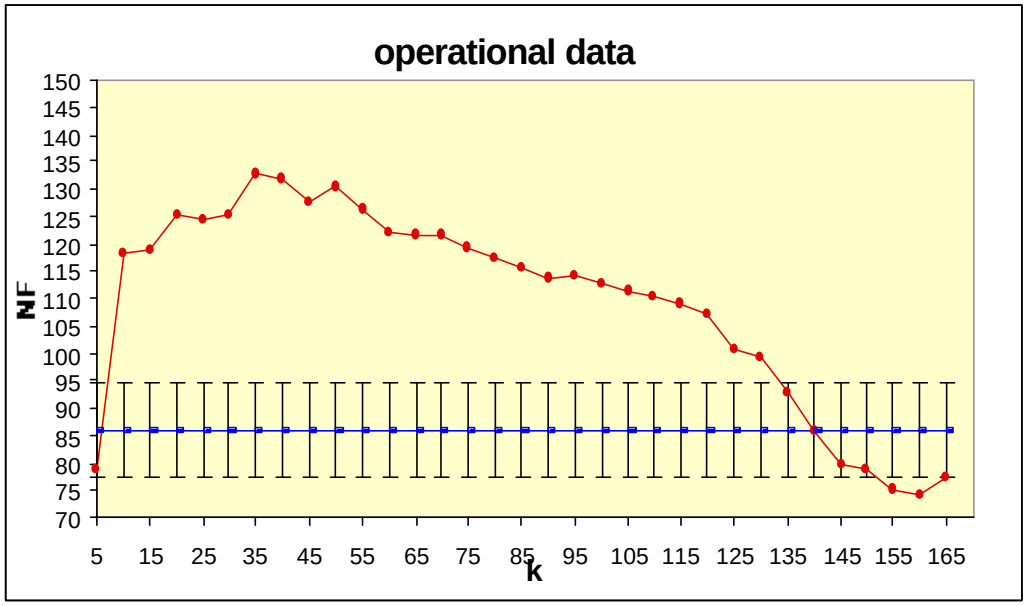


Figure 7 Prediction of the Two-Steps Bayesian model for operational testing

Summary

In this section we presented two dynamic methodologies, the One-Step and the Two-Steps Method, useful for predicting the cumulative number of failures at the end of the testing phase by using the data collected during the testing phase itself. The most attractive feature of these models is their simplicity: they only require collecting the time intervals between subsequent failures. No estimation of parameters of the product or of the development process is needed.

For prediction purposes in both the One-Step and the Two-Steps Method we used a Classical estimator and an alternative Bayesian estimator comparing their respective performances in the overall estimated number of failures by using a industrial cases study. As a result we conclude that the Bayesian approaches usually achieve better results with respect to the Classical also because the latter may require too much data to provide meaningful predictions.

Although we focused our attention on the non-operational test stage, due to the general nature of the proposed methods we also tried to apply them to operational data. For this reason we have reported here the results obtained by using an industrial case study.

7 Reliability Models Application

Preface

In this Chapter we continue the exploration of the testing phase, considering now its final stages, i.e. the operational testing. So far we have proposed only methods and approaches which can be applied: before beginning the testing phase, to schedule the time and effort required for its development (Chapter 4); till the release in operation of the final products, for deriving and prioritising the test cases (the Cow_Suite methods of Chapter 5) and for predicting the overall amount of failures discovered at the end of testing phase (Chapter 6). In particular, the latter models have been specifically created to deal with non-operational failure data even if some of them could represent a complementary approach for failure rate estimation.

Now we want to concentrate on the operational testing, which is the final test action prior to deploying the software. Thus by using the data collected in this stage, we want to apply the commonly-used methods (the reliability growth models) for evaluating some characteristics of the products as the level of reliability achieved.

In this Chapter we provide an introduction the Software Reliability Engineering (Sections 7.1) and a brief description both of the operational testing and the models applicable to reliability prediction (respectively Sections 7.2 and 7.3, 7.4). Finally, we report our experience in applying these last to a real case study provided by an industrial software developer (Section 7.5).

7.1 Software Reliability Engineering

Since the 1980s, with the increasing in the use of software systems in everyday life, a large part of software engineering has focused its attention on the quality of software components, and on the methodologies for controlling and evaluating them. In particular the range of the research included both the specification of a proper development process and the analysis of the software applications themselves. A “mature” and well-established development process can contribute to the quality of

its products but cannot completely guarantee about reliability level. Therefore the application of techniques for quantitatively evaluating and predicting the level of quality attained are necessary, especially in the situation in which it is crucial to confirm that a particular program has achieved its reliability goal before use. Often the developed systems are used in situations where failures and incorrect outputs can cause annoyance, loss of money, or even loss of human life. In literature there are several examples in which software failures caused dramatic disasters or killed people, such as the case of the Therac-25 radiation therapy machine [LI92]. These experiences have contributed to the birth and the diffusion of Software Reliability Engineering (SRE) which concentrates mainly on a very important software attribute: reliability. This is only one of the attributes of a software product and belongs to the specification of the general terms of dependability ([LA92], [LA93]).

Reliability, which is formally defined as *"the probability of failure-free software operation for a specified period of time in a specified environment*" ([AI91]), in particular represents quantitatively the "quality level" reached by a software product. Observing a program performing failure-free for a long enough period (or if a sufficiently small number of failures are observed during a long period of operation) it would be reasonable to accept claims that it is sufficiently reliable for its intended purpose. Unfortunately with some systems, specifically the critical ones, real operations cannot be used to obtain this kind of measurement and other methods for estimating their reliability must be used instead.

Before continuing it is important to specify that when referring to SRE the attention is focused not on faults but on failures i.e. the deviations of the delivered service from the functions for which the program was intended (Chapter 2). SRE is therefore not concerned with how many faults remain in the software product, as much as they are, but with how often the product will fail, and the impact such failures will have on the job they must do. However generally speaking about testing it is sometimes wrongly though that the more faults are identified, the greater the increase in reliability after their removal.

Unfortunately, the commonly used testing techniques stress the software in a different way with respect to the real uses to which the software will be subjected during operation, hence they do not provide a guarantee of the reliability attained. A classic example is reported by Adams [AD84]: he found that some of the 30% of the faults found in the system he studied would each show up less than once every 5000 years of operational use. Clearly, any testing procedure that was efficient at finding

these very “small” faults, but inefficient at finding “larger” ones (causing failures with higher frequencies), would not allow us to increase reliability efficiently. Thus conventional testing approaches can increase the reliability of the products, but they do not provide any final measure of what has been achieved.

In literature there are various works which try either to compare the reliability obtained by applying widely used testing techniques (for instance branch or data flow coverage (see Chapter 2)) and the methods based on the operational profile [ST97], [FHL98] [FY00], or to combine diverse software detection techniques to improve the final reliability [MZ98], [FO00], [LPS00], [LPS01a/b].

Regarding SRE, it is formally defined as the quantitative study of the operational behaviour of software-based systems with respect to user requirements concerning reliability [LY02]. SRE involves the entire process of development, from the feasibility to the maintenance phase, with the main intent of predicting, modelling, estimating and measuring the reliability of software products. In addition SRE defines the attributes of interest and the metrics applicable during the development process for measuring the products reliability, and specifies the different development phases as well as the system architecture.

Thus SRE attempts both to satisfy the customer’s needs and to define a proper plan and schedule of testing phase. The specification of precise reliability requirements facilitates the system tester, to verify that the developed system meets the requirements, and ensure to the customer the acquisition of the committed product. Moreover, the time dedicated to the testing phases is exactly what is needed for reaching the required reliability and the involved resources are only focused on the high-usage functions or operations avoiding energy waste. The main steps identified by SRE are:

a) Define and quantify product usage. Measuring reliability depends on the observation of the product behaviour in operation, and is strictly related to the environment in which the product will be inserted. In many cases, i.e. critical systems, the simulation of the product usage in the testing environment is necessary. This means specifying how the customer will use the various system features and which environmental conditions will influence the process. Testing by simulation of operational usage is known as *operational testing* or *statistical testing* or *software reliability engineering testing* (SRET). As will be specified in detail in the next section the key idea is that the selection of the test cases is carried out in such a way that the probabilities of selection are the same as those

in real operation. Consequently, any statistical estimation of reliability in the testing environment can also assumed to be as characteristics of operational reliability.

b) Define quantitatively the reliability goals with the customer. This is useful not only for providing the information necessary for demonstrating that the reliability requirement has been achieved, but also to maximize customer satisfaction and resolve any possible contractual controversy.

c) Track the reliability of the product during testing by executing the proper test cases. In operational testing the test cases are executed in random order, but based on usage probability. Hence those runs, which are of the greatest importance to the customer, are likely to occur more often.

d) Measure the reliability, i.e., interpret the test results obtained. There are two different testing behaviours as will be better discussed in the next section: testing to certify the reliability or testing to improve it. In the former case when failures are discovered during the testing phase they are left in the code (life testing) because the purpose is deciding whether the software is acceptable or not [ABK94]. In the latter the corresponding faults are removed, hence a sequence of programs is obtained showing (possibly) a growth in reliability which is in turn measured by the application of the Reliability Growth Models.

7.1.1 Achievable Reliability

In [PSM02] the authors analyzed the practical implications of varying probabilities of failure over input subdomains of operating regimes, and evaluated the possibility of estimating useful upper and lower bounds on the reliability of a two-versions system. Of course, it is very attractive for the customer requiring high reliability for the products he/she is going to acquire. Unfortunately the difficulty of achieving and demonstrating reliability is strictly related to the level of reliability required [LS00]: for instance if a reasonable failure probability is 10^{-3} per unit time, it is necessary to run at least 1000 test cases. Of course, the situation is more critical in case of safety critical systems developed for managing and controlling aircraft, industrial plants, railway and air traffic for which the reliability requirements must be very high. For instance, as reported in [LS93], in civil transport airplanes the quoted requirement is failure probability of at least 10^{-9} per hour of operation; in the U.S. Federal Aviation Administration's Advanced Automation System (for air traffic control), the required failure probability was at least 10^{-7}, i.e. 3 seconds per year.

These stringent requirements, called 'ultra-high reliabilities" [LS93], appear very difficult to reach and demonstrate by using available means: reliability growth models, testing with stable reliability, structural dependability modelling, as well as more informal arguments based on good engineering practice. In [LS93] and [LS00] the authors provided some rigorous arguments about the limits of what can be validated with each of such means. In particular they show that only combining evidence from these different sources it is possible to raise the levels that can be validated and reaching consequently ultra-high reliability. Recently alternative studies demonstrate that the use of Bayesian networks [MW82] can also be used for reaching this target [NFF03] and [NKF03].

The proof that high software reliability is attainable comes from earlier systems, which reach this level during extensive operational use. For instance the AT&T telephone system historically exhibited very high quality-of-service measures, achieved by focusing not only on component reliability [HMW01] but also an extensive redundancy, error detection and recovery capabilities.

Recently, with the increasing of the use of Component-Based software, the target of the literature has been partially oriented to the measurement and achievement of the overall reliability of an integrated software system [LRM97, LRM02, LHK02, YLK02]. Some interesting results are: [ST00] which focuses on the problem of component re-use, and [PO02] which treats an important problem in many safety-related industries, the reliability assessment of upgraded legacy systems.

As shown in this short, and not exhaustive overview of the literature, achieving an established level of reliability is not a trivial problem and many solutions have been provided over the years. Generally the topic of reliability is extensively treated, leading to the diffusion of the software reliability engineering practice in many fields. Examples of successes obtained are found in [MU03], which provides a complete list of published articles and papers, written by practitioners who have applied software reliability engineering to their projects and described their experiences resulting.

7.2 SRET

Software reliability is a measure of the probability that software will execute without failure for a specified time period within a specified environment. A key step in SRE practice is to quantitatively define the quality objectives of the development

process, which embraces all phases in the software life cycle, from the feasibility and requirement stages up to maintenance after delivery.

This section mainly focuses on the application of SRE activities to the testing stage. In mid-1990s, Musa introduced the Software Reliability Engineered Testing (SRET) methodology, whereby "SRET is testing guided by reliability objectives and expected usage and criticality of different operations in the field" [MU98].

In particular SRET can be applied in two different manners for debug testing and for acceptance testing as mentioned in the previous section. In the former, the metric estimated and tracked is failure intensity, i.e. failures per unit execution time. System testers use failure intensity to guide the bugs' correction process. Acceptance testing, on the other hand, does not involve fault removal to resolve failures, but enables an overall "accept" or "reject" decision.

In the following sections we briefly report the five principal activities of the SRET approach, referring the reader for more details to [LY96, MU93, MU96, MU98, MU03].

7.2.1 Defining the Reliability Objectives

As for the other requirements, the reliability must be specified in strict agreement with the customer. It provides the information necessary not only for later demonstrating that the requirements have been fulfilled, but also for resolving any possible contractual controversy, in the case of failure.

The first step for defining the reliability objective is to establish which operational modes need reliability verification. Specifically an operational mode is defined as a distinct pattern of system usage likely to stimulate different failures, or rarely-occurring failures with critical impact, and hence needing separate testing.

Many factors may contribute to individuate an operational mode, such as system maturity or overload, critical events and so on. [LY96, Chapter 5]. In the distinct modes of operation it is then necessary to define possible failures and identify their potential impact on users, i.e. the severity classes. Some classification criteria can be adopted for this purpose such as the human life impact, the cost impact as the loss of present or potential business or the service impact. Generally there are four levels of severity classes ordered in a decreasing manner and defined as:

Severity 1: Complete unavailability to users of essential services

Severity 2: Degraded availability to users of essential services

Severity 3: Unavailability to users of services, but workarounds available

Severity 4: Unavailability of capabilities that do not affect users

The last step is defining the reliability level to be achieved for each operational mode and severity class.

7.2.2 Developing the Operational Profile

The reliability of a product depends on how the costumer will use it; therefore the operational profile is the key notion in evaluating software reliability, and is what distinguishes operational testing form traditional debug testing.

Musa defines an operational profile as *"the set of operations and their probability of occurrence"* [LY96, Chapter 5], where an operation is a complete task performed by the system. To obtain the list of operations, a stepwise procedure can be followed.

- **First step:** the identification of different customer types, i.e. the initiators of operations. An operation can in fact be initiated by a user, a transaction, another system, or the system's controller, thus it is first of all necessary to group the users, who utilize the system in similar ways, into user types. These are then refined for identifying all the modes in which a user can invoke the system.
- **Second step:** the enumeration of the operations that are produced by each initiator. For each initiator, by using the documentation available such as system requirements, draft users manuals, a list of operation is produced. This will be further refined as requirements, design and implementation proceed. Specifically it could be the case either of dividing an operation into two or more, if the processing results substantially different in several cases, or reducing the number of operations for a less fine-grained representation of use.

  Sometimes, focusing only on testing purposes instead of defining the list of operations for each initiator, it is preferable to define the set of test inputs that will be used for exercising the software under test. A test input or an input point is defined as a set of values, one for each of the variables that affect the behaviour of the software under test.

  Hence an input point could take different forms depending on the software considered; for instance:

  - For a program, which receives all its input information at the beginning of its execution, the input variables form an input point, also called input vector. In this case all the possible values of an input vector form a set of input points.

- For a program having an internal state, which changes as consequence of a receiving input, each input point also includes in its variable the state variables with their initial values.

- **Third step:** determine the occurrence rates of each operation. For this purpose it is possible to either use existing field data from similar systems or to simulate the system behaviour, i.e. determine arrival rates of events that invoke different operations, or make estimates.
- **Fourth step:** determine occurrence probabilities by dividing occurrence rate by total occurrence rates. At this step a functional profile is defined, i.e. a list of the functions needed by the user in each mode and their occurrence probability.
- **Fifth step**: define the operational profile from the user's point of view. This must be converted to the operational profile, which is system oriented.

If in the second step the set of inputs has been defined, the operational profile consists in determining the probabilities of selection of the different inputs, and hence the fourth and fifth step are unified. Generally, the set of input points is very large (or even infinite); therefore it is not possible to enumerate each point with its probability. Three practical methods can be defined and possibly combined, for this purpose:

a) Specifying the probabilities as mathematical functions;
b) Subdividing the input space into a manageably small number of subsets, and specifying a list of probabilities, one for each subset;
c) Instead of specifying the probabilities, specifying the process for producing the input, if it exists, according with the intended distribution. This process could be a pseudo-random process, a simulation of the environment knowledge, the use of test operators, or of recorded input data set.

7.2.3 Preparing the Tests

In this step the test cases to be executed and the scripts for automatically launching them are prepared. In particular,

A test case is specified by an operation and its complete set of input variable values and environment. Once the set of test cases is established they must be drawn randomly following the operational profile using specific test procedures, i.e. the statistical specification of the set of runs associated with an operational mode, made by providing values of operation occurrence rates. The process for preparing test cases involves three steps:

- **First step:** Estimating the number of test cases, i.e., runs from the amount of testing scheduled and allocating them among the different operational modes. It is worth noting that the same test case can be executed in different test procedures and results in different test runs. Therefore the minimum number of test cases required is equal to the maximum number of runs allocated to an operational mode. In this manner in fact it is possible to avoid a useless waste of testing resources due to the duplication of runs. Executing the same run with exactly the same values for all input variables is only required for collecting more data, verifying that a failing run now operates successfully, or conducting a regression test.
- **Second step:** Specifying the test cases**,** i.e., selecting the operation with a probability equal to its occurrence probability in the operational profile and selecting the run with equal probability from all possible runs of the operation.
- **Third step:** Preparing the test procedure scripts, i.e. the procedure used for calling the test cases. Due to the generally large number of tests to be run the execution of the scripts should be automated whenever possible.

7.2.4 Executing the Tests

In operational testing, the test cases are executed in random order, but based on usage probability. Hence, those runs which are of the greatest importance to the customer are likely to occur more often. Statistical testing in particular depends on running large number of tests. Roughly for a reasonable chance of finding a fault in a program with failure probability *q*, or to demonstrate a failure probability of the order of *q*, it is necessary to run at least a small multiple of *1/q* test cases. For instance if a reasonable failure probability is 10^{-3} per unit time, it is necessary to run at least 1000 test cases.

Considering this order of test cases, manual testing becomes a labour-intensive activity. It requires a human tester to select the test cases that appear useful, running them through the software under test, and analysing the result for failure. Therefore the execution of test cases must be automated with the use of a test management system. This is responsible for setting up, executing test procedures scripts, capturing input and output, and cleaning up. Specifically it should implement a mechanism to automatically record execution parameters, results, failures, and their severity and time of occurrence. A set-up for automated testing is shown in Figure 1.

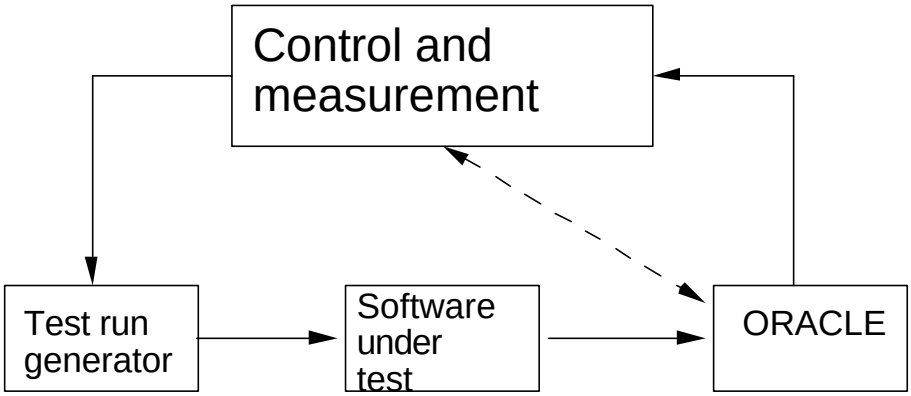


Figure 1 The testing environment

As shown by this figure, an important part of the test automation is the oracle. This name indicates any mechanism used to decide whether the program behaves correctly on a given test. The oracle decides about the test outcome for example by analysing the behaviour of the program against its specification. One of the main qualities of the oracle is having a high *coverage* defined as: the probability that the oracle rejects a test run, given the correct probability distribution of the inputs and given that the test run is a failure.

Having a good oracle is therefore an important and difficult to achieve objective in operational testing, because if failures pass undetected, software reliability cannot be increased and its assessment will be misleading. Also undesirable, even if less dangerous, are false alarms, because resources will in wasted to diagnosing inexistent failures. Here we do not extensively treat the problem or oracle definition referring the reader for more details to [MIO87, LY96 Chapter 5]. We limit ourselves to provide some useful guidelines for automatically checking the test results:

- **Specification checks.** The definition of a correct result comes from the specification by using assertion or formal executable language. For instance: for a program which has to find the solution of an equation, the check may consist of substituting the results back into the original equation.
- **Back-to-Back testing.** Any time a reference system is available, such as a previous version of the software, it is possible to compare its results with those produced by the program under test. The discrepancies may not always be due to bugs, as differences between the two versions may be allowed by the specification
- **Reversal checks.** When feasible, the oracle can compute the inverse function of the produced output and compare it back with the input.

- **Suspicious behaviours.** The oracle can be designed to look for illegal or unreasonable values of the variables, that are a necessary, although not sufficient, condition for failure.

7.2.5 Interpreting Test Results.

Failure data collected during test execution are interpreted differently depending on whether the objective is to resolve the detected failures or not, as in debug testing or acceptance testing. In the former case the goal is to increase product reliability; therefore during system testing, as the fault are removed, periodic estimates of failure intensity are made from failure data. Typically these estimates are computed by using a suitable reliability growth model, as will be described in the next section. The test stops when the reliability objectives are reached, i.e. for each severity class the established failure intensities are achieved.

The acceptance testing goal is to decide whether a product is acceptable or not. In this case the test consists in simply evaluating the product failure behaviour against the required reliability levels. Depending on the failure intensity achieved it is therefore possible to accept or reject the software being tested or continue testing.

Musa's SRET approach has been successfully applied to many projects with documented strong benefit/cost ratio results. As an example AT&T Bell Laboratories is currently applying SRET in a substantial number of communications software-based systems, and specifically in the Operations Technology Center of the Network Services Division, developers have used it on over 20 projects; the National Security Agency is now embracing the technology to build communications system software where security and reliability are logical and required ingredients.

7.3 Reliability Theory: Some Basic Definitions

In the previous section, some important concepts such as reliability and failure rate were mentioned without a rigorous definition. Before presenting the reliability growth modelling it is necessary to fill this gap by introducing key concepts, referring the reader to [Ly96 Appendix B] for more details. In reliability theory, interest is focused on a random variable *T*, representing the time to failure, and the probability that *T* is in some interval *(t, t+Dt)*:

Eq. (1) $P(t \leq T \leq t+\Delta t) \equiv$ probability that $t \leq T \leq t+\Delta t$

If *f(t)* indicates the density function and *F(t)* the distribution function the previous formula results:

Eq. (2) $P(t \leq T \leq t+\Delta t) = f(t)\Delta t = F(t+\Delta t) - F(t)$

Then considering that T is defined only for the interval 0 to $+\infty$ and

$$F(t) = \int_0^t f(x)dx$$

From Eq. (2) results:

Eq. (3) $$F(t) = P(0 \leq T \leq t) = \int_0^t f(x)dx$$

The *reliability function, R(t),* representing the probability of success at time t, is therefore defined as the probability that time to failure is larger than t (that is $T>t$), i.e.,

Eq. (4) $$R(t) = P(T > t) = 1 - F(t) = \int_t^\infty f(x)dx$$

In practice the density function *f(t)* is not used very much because the data observed are relative to the failures occurrences, and the *failure rate function* (or hazard function) is preferred instead. Explicitly the failure rate is defined as the probability that a failure per unit time occurs in the interval [*t, t+Dt*], given that a failure has not occurred before *t.* That is:

Failure rate≡ $$\frac{P(t \leq T < t + \Delta t \mid t > T)}{\Delta t} = \frac{P(t \leq T < t + \Delta t)}{\Delta t P(T > t)} = \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)}$$

The *hazard rate* is instead the instantaneous rate of failure at time *t*, given that the system survives up to t and is defined as the limit of the failure rate as the interval approaches zero, ($\Delta t \rightarrow 0$). That is:

Eq. (5) $$z(t) = \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} = \frac{f(t)}{R(t)}$$

As shown by these formulas, the function *f(t), F(t), R(t),* and *z(t)* can be transformed one into another. Just for instance, for any time *t* results:

Eq. (6) $$z(t) = \frac{dF(t)}{dt} = \frac{1}{R(t)}$$

.

Another important measure for characterizing a failure model using a single parameter is the *mean time to failure, (MTTF).* This is defined as the expected time during which the system will function successfully without maintenance or repair. That is:

Eq. (7) $$MTTF = E[T] = \int_0^\infty tf(t)dt = \int_0^\infty R(t)dt$$

where $f(t)$ is the density function and $R(t)$ is the reliability function.

Finally other important functions in reliability theory are the *failure intensity*, $l(t)$, and the mean value function for the cumulative number of failures $m(t)$. Supposing that $M(t)$ is a random process defining the cumulative number of failure by time t, $m(t)$ is defined as the mean value function, i.e.: $m(t)=E[M(t)]$

The failure intensity $l(t)$, is defined as the instantaneous rate of change of the expected number of failures with respect to time, and derives from $m(t)$ as derivative:

Eq. (8) $$l(t) = \frac{dm(t)}{dt} = \frac{d}{dt}(E[M(t)])$$

The software reliability theory is mainly based on the application of reliability growth models to evaluate reliability measures; it is therefore important to define exactly what this means.

Generally considering the successive interfailure times, T_1, T_2, …T_n, a growth in reliability can be experienced if the successive intervals tend to become larger, i.e. $T_i \leq T_j$, for all $i<j$. Precisely $P(T_i<v) \geq P(T_j<v)$ for $v>0$. Otherwise, if $F_{Ti}(x)$ is the cumulative distribution function of T_i the growth in reliability is represented by: $F_{Ti}(x) \geq F_{Ti}(x)$ for all $i<j$ and $x>0$ [ACL86].

7.4 Reliability Growth Models: an Overview

The software reliability models appeared for the first time in the 1970s with the pioneering works of [MJ72, SH72] and have been the most successful achievement in recent years for estimating and predicting the reliability of the systems. Nowadays there are dozens of these models, each one with its particular assumption. Basically the main characteristics of these models are [MIO87]:

- *Predictive validity:* the capability of the model to predict future failure behaviour from present and past failure behaviour (that is, data). This characteristic is significant only when some changes in the failure behaviour can be experienced,

as should occur when applying an operational test in which faults are repaired when detected.

- *Capability:* the ability of the model to estimate with satisfactory accuracy quantities of interest that could be for instance the present reliability, mean time to failure (MTTF), or the expected date of reaching a specified reliability.
- *Applicability:* the ability of a model to be applied in different conditions across software products that could vary in size, structure, or functions implemented. In particular, it is very important that a model is not dependent on development or operational environments.
- *Simplicity:* a model should be simple under three different aspects: in collecting data, which must be simple and inexpensive; in its concepts, i.e. anyone without extensive a mathematical background should be able to understand the model and its assumption; in its implementation, so that the model can be rapidly and inexpensively applied.

Indeed the reliability evaluation is a difficult task, mainly because it deals with software failures, caused by design faults, which reveal themselves only under appropriate operational circumstances. Generally, software reliability modelling is described as a set of techniques that apply probability theory and statistical analysis to predict software reliability, or also the modelling of past failure data for predicting the future system behaviour.

The approaches proposed are divided into two distinct categories, depending on the data used for the derivation of the stochastic process useful to reliability estimations: the number of failures discovered per time period, or the time between failures measured as wall clock or execution time.

It is worth noting that the accuracy of data collection can have a fundamental role in the reliability predictions obtained. Considering for example the first group, if during the development of the operational test for each failure only the date, and not the exact hour in which the failure occurs was reported, the unit of time will not be shorter than the day, and the reliability estimations will consequently have the same granularity. Moreover it is fundamental to consider only the period in which the operational test has been developed, otherwise the predictions should not be true. For instance, if the test is conducted only during working days at least Saturday and Sunday must be eliminated from the data collected

Referring to the second group, a peculiar situation is when the failure time is expressed in execution time, i.e., the failures are registered considering the time in

which the operational test is running on the CPU. In this case, the time between failures registered corresponds exactly to the period in which the operational test has run CPU. The same cannot be true when the clock time is collected. It is possible that the clock time between two successive failures is equal to one hour but due to internal problems the operational test has run on the CPU for only a few minutes.

Generally, with the proper information, the failure per time period data can be transformed into time between failures and vice versa. It is also possible to simulate this transformation by distributing the failures on the test period in a random or in a uniform manner.

Usually an informal description of the reliability growth modelling, start considering a program under debugging in which the execution time between successive failures, t_1. t_2. …t_{i-1}, can be experienced [ACL96, BCL90, BL92]. This represents the raw data that will be used for defining the stochastic process. In this perspective it is assumed that each time a failure is observed the fault causing it is fixed, but varying this hypothesis can derive different models [ACL96, MIO87]. The time between failures observed is thus considered to be realizations of random variables, T_1, T_2, …T_{i-1}, and used for predicting the future behaviour, T_i. T_{i+1}… (note that even the current reliability T_i results in a prediction).

For this purpose different *prediction systems* are developed and compared, where a prediction system is defined as [ACL96, BCL90, BL92]:

1. The *probabilistic model* which specifies the distribution of any subset of T_i's conditional on an (unknown) parameter a
2. a *statistical inference* procedure for α involving use of available data (realizations of T_i's)
3. a *prediction procedure* combining 1) and 2) for making probability statements about future T_i's

It is worth noting that having a "good" model is not enough for truthful predictions; the three points mentioned are strictly related. For instance, in the models considered the random variables used are not identically distributed, therefore it is not possible to analyse the models' performance with the traditional "goodness-of fit" method [LO87, HSM02]. Moreover, generally the representation of the software engineering process is a difficult task and requires complex models; therefore, it is not possible to choose a priori in favour of one model instead of another. The selection of the model, which better represents the software engineering process requires an accurate analysis, as will be explained later in this section.

In the following we present a brief summary of the software reliability growth modelling, referring to [LY96] for a more detailed documentation.

7.4.1 Model Classification

To simplify the model selection, and in particular an accurate model organization, Musa and Okumoto [MU83] present a model classification schema, based on the different attributes of each model which are:

1. *Time domain.* Wall clock versus execution time
2. *Category.* This represents the total number of failures that can be experienced in infinite time, which can be either finite or infinite.
3. *Type*. The distribution of the number of the failures experienced by time *t* is considered
4. *Class.* This attribute is characteristic only for the finite failure category. This expresses the functional form of the failure intensity expressed in terms of time.
5. *Family.* This attribute is characteristic only for the infinite failure category. This express the functional form of the failure intensity expressed in terms of the expected number of failures experienced.

Generally the attributes considered most frequently for grouping the different models are category and type. Considering the former, if $m(t)$ indicates the mean value function for the cumulative number of failures, as defined in the previous section, the models are divided into two basic groups, depending on $m(t)$:

- the finite failure models subgroup if $\lim_{t\to\infty} m(t) < \infty$
- the infinite failure models subgroup otherwise.

The type attribute instead categorizes the types of data the model uses. In particular two subsets can be individuated, even if there are models that can handle either groups.

First group: This is represented by all models that use as input data the observed number of failures discovered per time period. Generally distribution of the number of the failures experienced by time *t* is considered to be the Poisson type [HSM02]. In this case if the interval $[0,t]$ is divided into *n* partition such that $t_0=0, t_1,\ldots t_n=t$ and $m(t)$ indicates the mean value function for the cumulative number of failures, each of the f_i independent Poisson random variables has:

- mean value $E[f_i] = m(t_i)-m(t_{i-1})$
- probability density function $P(f_i = x) = e^{-(m(t_i)-m(t_{i-1}))} \frac{[(m(t_i)-m(t_{i-1})]^x}{x!}$ for x=0,1…

Second group: This is characterized by the models that use as input data the observed time (actual wall clock or some measures of computer execution time) between software failures. In this case the distribution of the number of the failures experienced by time *t* is considered of the Binomial type [HSM02].
Combining the two attributes a classification of some of the existing reliability models can be made, as shown in the table below.

Failure per time period		**Time between failures**	
Finite failure category	**Infinite failure category**	**Finite failure category**	**Infinite failure category**
Books and Motley[BM80]	Duane model [DU64]	Jelinski and Moranda [MJ72]	Geometric [MO75]
Non-homogeneous Poisson process [GO79]	Musa-Okumoto Logarithmic Poisson [MIO87]	Non-homogeneous Poisson process [GO79]	Musa-Okumoto Logarithmic Poisson [MIO87]
Schneidewind's model [SC75]		Musa basic [MIO87]	Littlewood and Verrall [LV73]
Yamada S-Shaped [YOO83]			
Hyperexponential model [OH84]			

Table 1 Classification of the Reliability Growth Models

7.4.2 Reliability Growth Model Selection

Every reliability growth model (RGM) is characterized by accurate applicability conditions and a mathematical specification. In this section we mainly concentrate on the former aspect referring the reader to [LY96, Chapter 3] for an exhaustive mathematical description of these models. In particular, we report the three common assumptions, generally called *Standard Assumptions*, which are fairly standard for each of them. They are considered fundamental preconditions for applying every model.

A1. The software is operated in a manner similar to that in which reliability predictions are to be made.

A2. Every failure within a severity class has the same chance of being encountered as any other in that class

A3. The failures, when the faults are detected, are independent.

The first assumption ensures that data used for deriving model estimates are collected in an environment that can be assumed equal to that in which the reliability projections are to be made. Any significant discrepancy between the two environments could in fact invalidate the predictions obtained. The second assures that different severity classes may have diverse failure rates, and therefore require a separate reliability analysis, but the failures in the same severity class must have the same distributional property. The latter assumption allows simplicity in deriving the estimates, as the maximum likely estimates [HSM02].

A major step in predicting software reliability is to decide which model is the best for predicting reliability in a specific context, since there is not a model which performs better than the others in any case.

The procedural process for deciding which RGM should be applied for obtaining truthful reliability prediction has three main steps:

First step: Verify the Standard Assumption at least for A1 and A2. These are fundamental prerequisites for applying any model.

Second Step: Selecting the set of RGMs that can be applied. On the bases of the typology of the data collected, i.e. failures per time period or time between failures, a first choice of the models applicable for obtaining prediction can be derived, as reported in Table 1.

Third Step: Evaluating the specific requirements of any model. Among the models chosen in step two, a refined selection can be performed by analysing their specific assumptions or mathematical definition. For instance some models assume either that the number of faults in the software has an upper bound, or a perfect debugging. If testing is performed when the software is still immature, i.e., it is still possible to make many and significant changes to the software under test, it would be more appropriate to choose those models that do not assume an upper bound to the number of faults (e.g. Musa-Okumoto Logarithmic Poisson model or Littlewood-Verrall model). Otherwise, it would be more appropriate to choose a model that belongs to the finite failures category (e.g. Non-homogeneous Poisson Process (NHPP) model or Musa basic model). Moreover, if previous experience on similar projects indicates that a significant number of repairs results in new faults being inserted into the software, it would be more appropriate to choose from those models that do not assume perfect debugging (e.g. Littlewood-Verrall model). Otherwise, it

would be more appropriate to choose a model such as the NHPP model or Musa Logarithmic Poisson model.

Therefore the RGM can be differentiated by the assumption regarding testing and defect repair, even if, it is important again to stress that currently there is not a known method for determining a priori which model will prove optimal for a particular development effort. To this purpose in [WO97] the author tries to evaluate the effect of violation of the models' assumption on the inaccuracy of the model predictions.

Fortunately, recent theoretical advances in the field have largely eliminated some difficulties that arose in the choice of these models for a specific case. As will be better described in the next section, in fact, from the practitioner's perspective several software reliability tools are available today which facilitate the automatic execution of Step 3. These tools help users to make the choice of a model without requiring extensive knowledge of the mathematical aspects of the RGM. In particular, they let the user readily apply the best known software reliability models to his/her set of failure data and then choose the model that gives the best predictions, by analysing the produced results.

7.4.3 Survey of Reliability Estimation Tools

Generally the tools for reliability estimation by using the results of operational testing are divided into three categories: the steady-state reliability estimation, which is mainly based on the Markov model [KA86] and represents the system behaviour as a set of mutually exclusive system states, the modelling of system reliability based on component reliability, in which the system is decomposed into functional entities consisting of units or subsystems, and the reliability growth models, which are those presented in this section.

For the first two categories many tools exist that were developed outside the software domain and can be conveniently be used for modelling software reliability as well. Without claiming to be exhaustive, here we extend and update the survey of the available tool for reliability estimation presented in Lyu's book [LY96, Appendix A], accompanying each of them by brief description.

- **SMERFS** (Statistical Modelling and Estimation of Reliability Functions for Software) a portable (written in FORTRAN 77), menu-driven tool developed by the U.S. Naval Surface Weapons Centre. The tool provides the statistical modelling estimation of reliability function [SME96].

- **CASRE** (Computer Aided Software Reliability Estimation) developed at JPL which is a PC-based extension of SMERFS meant to provide a more user-friendly interface, more input data manipulation capabilities and the possibility of combining models to obtain different models which may provide better predictions [CAS00].
- **SoRel** developed at LAAS-CNRS in Toulouse, a Macintosh-based program with features similar to SMERFS, and also providing trend tests [SOR93], [KKL97].
- **AT&T Software Reliability Engineering Toolkit** developed for the first time by AT&T in1977 is a MS/DOS or UNIX® operating system implementing the Musa basic and the Musa/Okumoto logarithmic Poisson execution time software reliability models [LY96].
- **SRMP** (Statistical Modelling and Reliability Program) developed in 1988by the Reliability and Statistical Consultants, Limited of the United Kingdom, is a command-line-oriented tool for an IBM PC/AT or UNIX® operating system with characteristic similar to SMERFS [LY96].
- **ESTM** (Economic Stop Testing Model Tool) implemented by ESTM System in 1992 in a command-line-driven system for UNIX® operating system, that can be used to help decide when to stop testing [LY96].
- **M-élopée** developed in the 1996 by Mathix for France Telecom is a PC-based tool which facilitates the direct import of input data and their manipulation. It implements several reliability growth models and the trend test [MEL].
- **FRestimate** developed by SofRel Company 2000, Sugar Land, Texas is a PC-based tool which allows the prediction of software defects, failure rate, MTTF and reliability before the testing. It also has estimation models to be applied during testing for reliability prediction [FR02].
- **Goel-Okumoto Software Reliability Model:** This is an automated version of the Goel-Okumoto Nonhomogeneous Poisson Process Software Reliability Model which runs on an IBM-PC or compatible under MS-DOS 2.11 or higher distributed by DACS (Data & Analysis Center for Software) Rome, NY. This tool also provides the Kolmogorov-Smirnov statistic and different estimations about fault and failure [GOEL].
- **Reliability & Maintenance Analyst** distributed by the Engineered Software, Inc. Belleville, MI This is a PC-based reliability analysis software package which permits the life data analysis [REMA].

- **PRISM** is an IBM PC, or compatible, Reliability Analysis Center (RAC) software tool that links together several tools into a comprehensive system reliability prediction methodology. It permits combining together different factors that influence system reliability [PRIS].
- **SREPT** Software Reliability Estimation and Prediction Tool. This is a unified framework containing techniques (including the architecture-based approach) to assist in the evaluation of software reliability in all phases of the software life-cycle [RGK00].

Here we only discuss the characteristics of CASRE and SoRel, which are those applied in the next section with the case study considered. The main result the tools calculate is product reliability (present reliability as well as future predictions of reliability) as a function of test time. For this purpose the tools allow the analyst to choose the model parameter estimation method: either maximum likelihood or least squares [HSM02]. In particular the tool CASRE can represent the product's reliability in terms of several interrelated reliability measures, such as cumulative number of failures, failures per time interval, and the product's reliability function.

Moreover, for each model the tools CASRE automatically compute the prequential likelihood function (PL) [BL92], which is a general means of comparing the accuracy of predictions provided by more models when applied to the same failure data set. In particular, the best model is characterised by the highest value of PL (we must specify that for convenience many times the tools compute -ln(PL), so in this case the best model is that showing the lowest value).

Regarding the tool SoRel, we must specify that it is the only one that allows several reliability trend tests such as: the arithmetical test, the Laplace test, the Kendall test, and the Spearman test [KKL97]. These tests allow an analyst to identify whether the reliability function is increasing, so that an appropriate model can be applied.

7.4.4 Using the Tools for Predictions

In this section we describe a stepwise procedure useful for predicting the reliability of software starting from a set of failure data collected during the operational testing. In particular we details the use of two tools, CASRE [LN92] and SoRel [KKL93].

For simplicity's sake as input data set, we consider only the observed number of failures discovered per time period (The steps considering as input data the observed

time between failures are similar). This allows immediately a preliminary selection among models that can be applied for prediction. Specifically, the reliability growth models usable with the tool CASRE are only: Brooks and Motley model, Generalized Poisson model, Non-homogeneous Poisson model, Schneidewind model, Yamada S-Shaped reliability model.

7.4.4.1 First Step: Applying SoRel

A basic assumption for using reliability growth models is that the failure data exhibits a growth in reliability; otherwise is quite difficult to obtain useful predictions [KKL97]. The first step is therefore to verify if the set of data available has this property, i.e. verify their trend. Generally there are two typologies of tests, called *trend tests*, applicable: the graphical test or the analytical test.

The former is very informal and consists in plotting the failure data, interfailure time or number of failures per unit of time, versus time in order to visualize the trend displayed. The latter has as principle the test of null hypothesis H_0 versus an alternative H_1 [HSM02]. Usually H_0 is assumed that the process is a homogeneous Poisson process and H_1 that the process undergoes monotonic trend, that is to say that the data exhibit only an increasing (decreasing) interfailure times or decreasing (increasing) failure intensity.

Regarding the analytical tests previous studies have shown the Laplace test to be optimal within the framework of the most famous software reliability models [GA92]. Briefly, this test consists in computing the *Laplace factors u(i),* expressed according to the failure data available (interfailure times or number of failures per time period) for the observation period *[0, T]* (for further details we refer to [KKL97]). In particular dividing the interval *[0, T]* into *k* units of time of equal length and letting *n(i)* the number of failures observed during time unit, it is possible to calculate the Laplace factors as:

Eq. (9)
$$u(k) = \frac{\sum_{i=1}^{k}(i-1)n(i) - \frac{(k-1)}{2}\sum_{i=1}^{k} n(i)}{\sqrt{\frac{k^2-1}{12}\sum_{i=1}^{k} n(i)}}$$

The meaning of these values is:

a) A negative *u(k)* suggests an overall increase in reliability between data item *1* and data item *k* and thus decreasing failure intensity.

b) a positive *u(k)*suggests an overall decrease in reliability between items *1* and *k* and thus and increase in failure intensity.

c) Values oscillating between -2 and +2 indicate a stable reliability

The tool SoRel permits the automatic derivation of the Laplace factors displaying the numerical results and the visualizing the corresponding curves. This shows the trend over a given interval of time - the global trend – highlighting the regions in which there is or is not an increase in reliability in the failure data. In particular the graphical representation can illustrate information about the local behaviour of the failure intensity, represented by the region in which there is a change in the trend. In fact the intervals in which there are decreasing Laplace factors indicate a local decrease in failure intensity or local increase in reliability. Intervals in which there are increasing Laplace factors suggest a local increase in failure intensity or a local decrease in reliability.

Considering that the reliability growth models may be applied when the data exhibits a growth in reliability, the local behaviour is useful for choosing the proper time origin or end within the global observation period, so that the subinterval of data considered has this property. The change in the time origin does not result in a simple translation. It is important to specify that sometimes it is possible to apply several reliability growth models even if there is a stable reliability in the failure data, but this strictly depends on the data available. In the next section we present an application of a real case study.

In the literature, studies for relaxing the assumption of a growth in reliability for the application of the RGMs have been performed. In Chapter 6 we also face this problem, proposing a Bayesian model for predicting the final number of failures at the end of the testing phase. The impossibility of RGMs usage is a critical problem mainly in the early phase of testing when the process of failure detection is not fairly stable and a large number of failures are experienced. In these conditions it is very difficult to experience a growth in reliability. Some research solutions are found in [KM91] in which the authors predict the number of failures occurring during a finite future time interval from the number of failures observed during an initial period of usage by using the RGMs; [XHW97] in which an approach using information from similar projects in order to obtain an early estimation of the model parameter for a current project is studied; [MD97] in which the authors estimate priori values that can serve as a check for the values computed at the beginning of testing, when the test data is dominated by short-term noise.

7.4.4.2 Second Step: Models Running

The second step consists in deciding which model is the best one for predicting reliability, considering the particular context. Sometimes it is possible to discard a model a priori, but generally is easier to apply all the models available to the failure data with the help of the tools, and then decide which is the best. In our case, we adopted the tool CASRE reporting here the main actions for using it. Further details are in [LN92].

1. *Environment set:* Before applying the models to the failure data it is necessary to set up the environment that consists in:
    a. Choosing the maximum-likelihood or least-squares parameter estimation. For maximum likelihood estimation, the parameter estimates are such that the value of the joint probability density of *n* random variables (called the likelihood function) is maximized. For least squares, regression techniques are used to find estimated values of model parameters.
    b. Specify the range of failure data over which software reliability models will be applied. On the bases of the Laplace trend test it is possible to discard failure data belonging to an initial or a final period (or both) and thus specify the proper data range that will be used as input to the models. The default data range is the entire data set.
    c. State clearly the interval over which the initial estimates of model parameters will be made. By default, the first half of the entire data set is used to make the initial estimates.
    d. Specify for how long the models have to predict the failure counts (for example for the next k>0 test intervals after the last point in the data range used as input to the models)
2. *Models running*: The models used for software reliability estimation may be chosen and run using currently-open failure data as model input. The model results are displayed in the graphical display window for analysis. In particular for each model, different views can be derived:
    a. Time between failures: for the failure count data, the time between failures for each test interval is taken to be the length of the test interval divided by the number of failures in that interval.
    b. Failure counts: the failure data is in the form of failure counts and test interval lengths. For the raw failure data, this display shows the number of failures encountered during each test interval. When showing model

results in this way, the plot shows the number of failures the model estimates in each test interval, as well as the number of failures the model predicts will appear in the future test interval.

c. Failure intensity: For failure data and models results, this plot shows the number of failures per unit time.
d. Cumulative failures: the plot shows the cumulative number of failures as a function of test interval number.
e. Reliability: this display shows the reliability indicated by the model results. For each observation in the raw failure data, a hazard rate can be computed. If we call the hazard rate h and the time interval is t, the reliability of the software is given as e^{-ht}. The plot shows the way in which the software reliability changes as failures are observed and faults are corrected.

7.4.4.3 Third Step: Models Selection

So far the different reliability growth models have to be run to obtain specific results, now the one providing the best estimation must chosen. For this, some mathematical analyses can be performed.

The first is the *goodness of fit test*, which is useful for deciding when a theoretical distribution can be used to correctly represent a given empirical distribution. The most popular are the *Chi-square test* and the Kolmogorov-Smirnov test. The former is applied when the failure data are relative to discrete random variables, as in the case of number of failures per time period, the latter to continuous random variables, as for the interfailure time. We briefly report here only the details of the Chi-Square test, referring to [LY96], [GA99] for more details. This test assumes that the distribution considered can be approximated by a multinomial distribution. In particular considering X the r.v. with distribution $F(x)$, such that $p_i = F(x_i) - F(x_{i-1})$ is the probability that a failure is in the interval $[x_{i-1}, x_i]$, n the number of observations, N_i the observed number of times that the measured value of X takes value i (i.e. a binomial variable with parameters n and p_i), the Chi-square can be calculated as:

Eq. (10) $$c_{df}^2 = \sum_{i=1}^{k} \frac{(N_i - np_i)^2}{np_i}$$

where df are the degrees of freedom, i.e. if k is the number of possible values for the variable X then generally $df = k - 1$. In particular, for the approximation to be

accurate, each $n \cdot p_i$ value should be moderately large ($n \cdot p_i$ = 5) otherwise some observations must be combined. In such a case if h is the number of grouped observations, $df = k - h - 1$. This value can be measured in terms of statistical significance if χ^2_{df} it is seen as the value of a random variable X_{df}. The boundary value, or critical value, of the acceptable range $c^2_a(df)$ is chosen such that: $P(X_{df} > c^2_a(df)) = a$ where a is called *significance level* of test. Thus the null hypothesis H_0 is rejected if $\chi^2_{df} > c^2_a(df)$. Usually a is chosen to be 0.05 or 0.1.

Applying the tool CASRE a table of goodness of fit statistics for all models that have been run is automatically displayed. These values are used in conjunction with the table of percentage points for the Chi-Square distribution to determine the significance level at which the model estimates fit the data. The models for which the Chi-Square value is not within the specified interval are discarded. Specifically the significance level is identified considering the degrees of freedom that the tool CASRE has calculated for each model (df), and finding in the table of percentage points values x_1 and x_2, which have the coordinates $x_1 = (df, 0{,}05)$ and $x_2 = (df, 0{,}95)$ that correspond to a significance level is 0.95%. Hence all the models for which the Chi-Square statistic is not within this interval $[x_1, x_2]$ are not considered.

The best model is finally chosen from those that have passed the Chi-Square test, considering other evaluations, the *prequential likelihood (PL)* and *the prequential likelihood ratio (PLR)* [BL92]. Briefly, if n is the number of observations, the former is calculated considering the cumulative distribution $F_i(t)$, derived for the reliability prediction by using the first $j-1$ failure data collected, $j=1\ldots i-1$, and in particular the correspondingly probability density function $f_i(t)$ with the formulas:

Eq. (11) $$PL_n = \prod_{i=j+1}^{n} f_i(t_i)$$

The prequential likelihood function evaluates the differences from the sequence of predictor densities and the true (unknown) distribution of the failure data, also revealing the situation in which the prediction values are very noisy even if their expectation is close to the true one. In both cases the PL assumes small values. It is worth noting that since PL tends to become very small the natural logarithm of the PL is calculated instead. Hence the best model is characterised by the highest value on PL or the lowest $-ln(PL)$.

The prequential likelihood ratio is used instead to compare the performance of the different models. Two prediction systems *A* and *B* can be evaluated via their prequential likelihood as:

Eq. (12) $$PLR_n = \frac{\prod_{i=j+1}^{n} f_i^A(t_i)}{\prod_{i=j+1}^{n} f_i^B(t_i)}$$

In this case if $PLR \to 8$ as $n \to 8$ then the system B is discredited in favour of A; if PLR exhibits neither upward or downward trends nothing can be said about the superiority of one system over the other.

The tool CASRE automatically calculates the PL for the different models, displaying in a plot the way a model's prequential likelihood statistic (CASRE computes *-ln(PL)*) changes with time. Moreover, given two models, CASRE also computes their prequential likelihood ratio useful for how much more likely it is that one model will produce more accurate estimates than the other.

In the next section an application of this evaluation to a real case study is presented. In the case in which the failure data are relative to the interfailure time other estimations can be used for choosing the best model such as the U-plot and the Y-plot [BL92], [CAG99].

Briefly, the U-plot is a general technique for detecting systematic objective differences between predicted *(Fi(t))* and observed failure behaviour. If the predictor system is working well the u_i's ($u_i = F_i(t)$) look like a random sample of the uniform distribution on (0,1), U(0,1). In particular this verification is performed plotting the sample cumulative distribution function *(cdf)* of the u_i' s and the cdf of U(0,1): if the U-plot is above the line of unit slope then the predictions are too optimistic, otherwise the predictions are too pessimistic.

Instead the Y-plot is useful for examining the u_i' s trend. For this purpose the KS-distance analysis is used which measures the max vertical deviation from the line of unit slope to the Y-plot. If the KS-distance is not statistically significant the errors in the predictions are in some sense stationary. In this case it might be possible to correct the failure predictions (Recalibration Technique) [LY96 Chapter 4], [BCL90], [BLA98]. If the predictions show an evident and constant bias is possible to use this information to recalibrate the future raw prediction in order to eliminate such errors.

Briefly, the steps for the recalibration procedure are:

1. Analyze the U-plot and verify if there is an (approximately) constant relationship between prediction and through
2. Obtain the raw prediction at step *i* of the distribution of the time to next failure
3. Calculate the recalibrated prediction as discussed in detail in [LY96]
4. Repeat this at each stage *i.* In this way a sequence of recalibrated predictions will result.

Even if the CASRE do not implement the recalibration procedure, in literature there are successful examples of the application of these techniques [BCL90], [BLA98] so that the same authors also suggest applying the recalibration to all software reliability predictions.

7.5 The Application Results

Here we present the application of reliability growth modelling within the context of Ericsson Lab Italy (ERI in the following), with the intention of improving the capabilities of this organization in statistical process control and in prediction methods [BLM98]. In particular, the objective is reducing below a determined value the *fault density* figures, that are obtained by monitoring the first six months of operation of released products. Recalling what was explained in Chapter 6, Section 6.6, the fault density in the ERI context is measured by the ratio between the cumulative number of failures observed in that period over the product size, expressed in lines of code. Root Cause Analysis (RCA) of reported failures is routinely performed, to track down failures to the phase in which they originated.

In this section we focused on reducing the fault density figures in the Function Test phase (Chapter 6) by introducing explicit reliability objectives to guide it. For this, a baseline project has been selected for applying reliability testing techniques, as described in Section 7.2 and performing the test selection so as to reproduce the expected usage of the system in operation.

7.5.1 Case Study

The project used as a base to evaluate the SRET approach (Section 7.2) is the ‘CTM project’. The CTM project implements the service Cordless Terminal Mobility (CTM) in the Ericsson AXE architecture. CTM is a service that allows users of cordless terminals to be mobile within and between networks. Where radio coverage

is provided and the cordless terminal has appropriate access rights, the user will be able to make calls from, and to receive calls at, any location within the fixed public and/or private networks, and to move without interruption of a call in progress. The solution adopted by Ericsson is to connect the mobile terminal to the fixed network via a Central Control Fixed Part (CCFP), whose main aim is to concentrate the traffic towards the CTM Exchange in a more capacious link called a 'device'. All the devices between a CCFP and a CTM Exchange are grouped, for administrative reasons, in an entity called a 'route'. The Ericsson CTM architecture is shown in Figure 2. The baseline project consists of the administration functionalities of the links between CCFP and CTM Exchange.

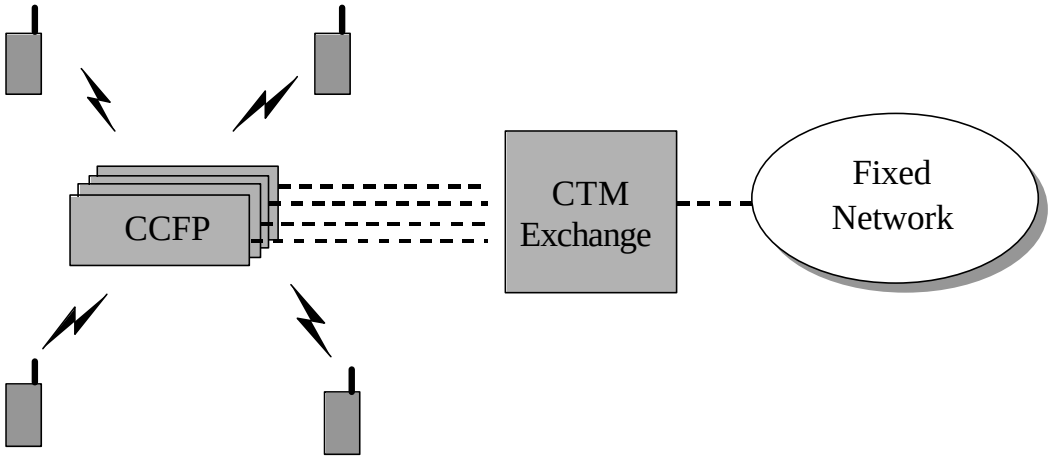


Figure 2 Ericsson CTM architecture

The steps of the Musa's SRET approach have been applied to the baseline project as described below:

1. Definition of the required reliability. The ERI improvement objectives require that fault density, i.e., the number of the failures found in the first six months of operation over the product size, is reduced to less than 0.15. We have estimated that only 1% of the activity of a CTM-AXE10 switching is spent for administrative functions, which are those in the baseline project. From this, we estimated that the reliability required for the system under test is on the order of 10^{-3} failures per hour of operation.
2. Definition of the operational profile. Actually this is because never before had reliability modelling been attempted within ERI. Consequently on some occasions during the implementation of the SRET process we found ourselves in role of pioneers, having to decide on acceptable engineering compromises, where

part of the information or the background required was missed. In particular operator manuals and previous experiences in similar systems were used as input to derive the operational profile. In the following tables we describe the results of the profiles obtained, according to the SRET approach. Regarding the operational profile, we used an implicit approach, i.e., the profile is represented by a behavioural tree. In Figure 3 we show only one branch of this tree. The severity of the failures is considered when assigning the probabilities.

3. Test case definition. We used the test instructions prepared according to the standard Function Test process
4. Set-up of the test environment. We integrated a proprietary Ericsson tool, called 'AUTOSIS', with an ad-hoc developed tool 'STUT'. AUTOSIS is a tool used for testing any system provided with an interface for man-machine interaction. The test is performed by sending commands towards the test object and by analysing the printouts obtained. The instructions to perform the test are supplied in the form of AUTOSIS test instructions (TIs), which are written prior to execution. A TI contains AUTOSIS instructions interpretable by AUTOSIS and commentary text. The tool generates two output files:
   a. A log file, with the execution trace, to be used in the debugging of errors;
   b. A report file, recording the execution time and the result of each test case.

   AUTOSIS has been extended in order to handle random variables with the integration of STUT. STUT is a tool that was developed to select the test cases according to the SRET approach. It generates a file specifying the test instructions to be used as input to AUTOSIS by processing the following input information:
   - The operational profile and the related test instructions for each leaf of the tree;
   - The definitions of the random variables;
   - The required reliability value.

Customers profile	
Customer	**Probability**
Telecom Operator	100%
Users profile	
User	**Probability**
Operator	100%
System-modes profile	
Mode	**Probability**
Installation	10%
Operation	90%

Functional profile		
Function	**Installation-mode Probability**	**Operation-mode Probability**
Definition of a route	26,2%	14%
Deletion of a route	0,78%	7%
Change of route parameters	1,56%	14%
Print of route parameters	31,4%	21%
Connect a device to a route	28,8%	17%
Disconnect a device from a route	0,78%	7%
Print device data	5,24%	10%
Print device connections data	5,24%	10%

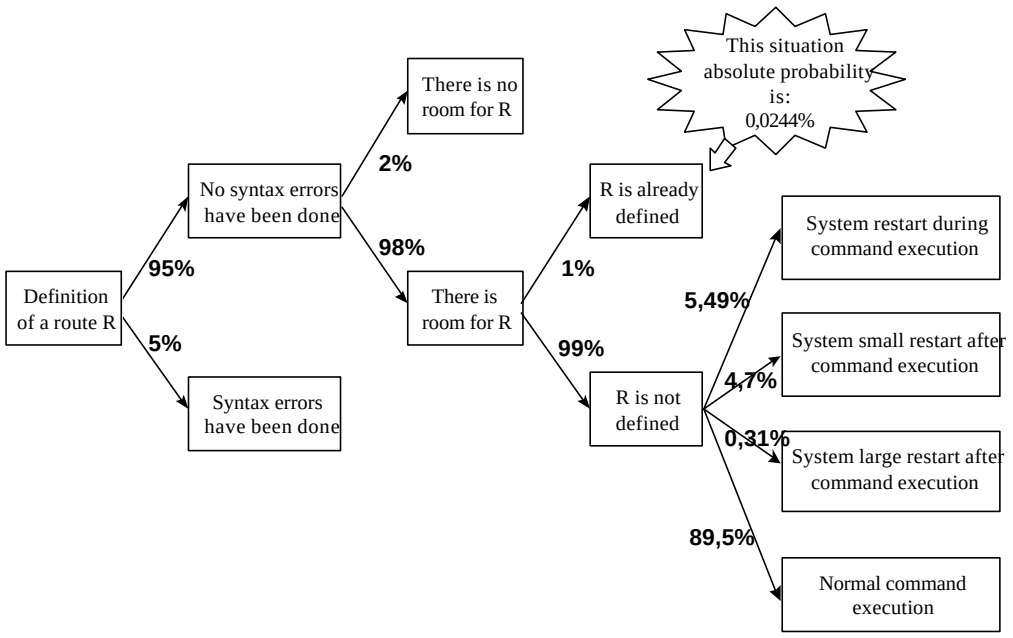


Figure 3 Operational profile for 'Definition of a route' in the ' Installation' system-mode

The use of STUT does not require any specific training for ERI testers, because the structure of the input file is quite similar to the structure of the AUTOSIS input file.

5. Execute the test cases. To execute test cases the following steps were performed:
   a. STUT, for the selection of a set of test cases;
   b. AUTOSIS, for executing the selected set of test cases: each time a failure is observed, we fix it, update the reliability of the system under test and return to step a).

   This process was repeated until the required reliability was reached.
6. Evaluation of reliability. ERI monitors and logs all failures found in the field for the first six months of operations. At present, such data are not used directly for reliability estimation; on the contrary, they are used to evaluate the failure densities, and the timing of failures is thus not directly accessible. However, we got access to the detailed failure report, including timing information, for a completed product which is similar to the baseline product. To these data, we applied standard techniques for reliability evaluation; the findings of this analysis are summarized in the next two subsections.

7.5.2 Data Analysis

As described in Section 7.4.4.1, as a first step we performed a detailed analysis of the data available, in order to verify their suitability for applying reliability growth models. The very first assumption for reliability assessment is that the analysed product is exercised according to the operational profile. This is not the case for the Basic, Integration and Function Test phases (at least, not so far). Thus, for the product examined, only the failure data relative to the six months of monitored operation are meaningful. For these six months, we had a reported number of 35 failures, collected over a set of five installations run in parallel. The products run continuously and failures are reported on a daily basis. For this reason, and also to comply with corporate established attitudes, we opted reliability growth models in the class of number of failures per time period. The period to be considered as the time unit naturally corresponded to one calendar day, in turn corresponding to five days of execution time, by considering the five plants monitored.

Another very basic assumption for using reliability growth models is that the failure data exhibit a growth in reliability. For this we applied the Laplace test with

help of the tool SoRel, and somewhat surprisingly, this was not the case if we considered the data set as a whole as shown in Figure 4.

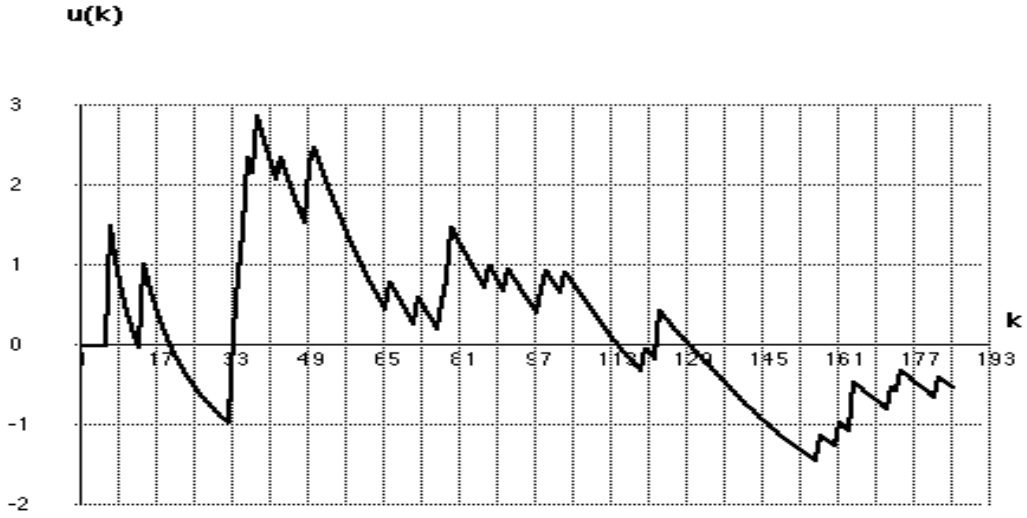


Figure 4 The Laplace test considered in the overall set of failure data

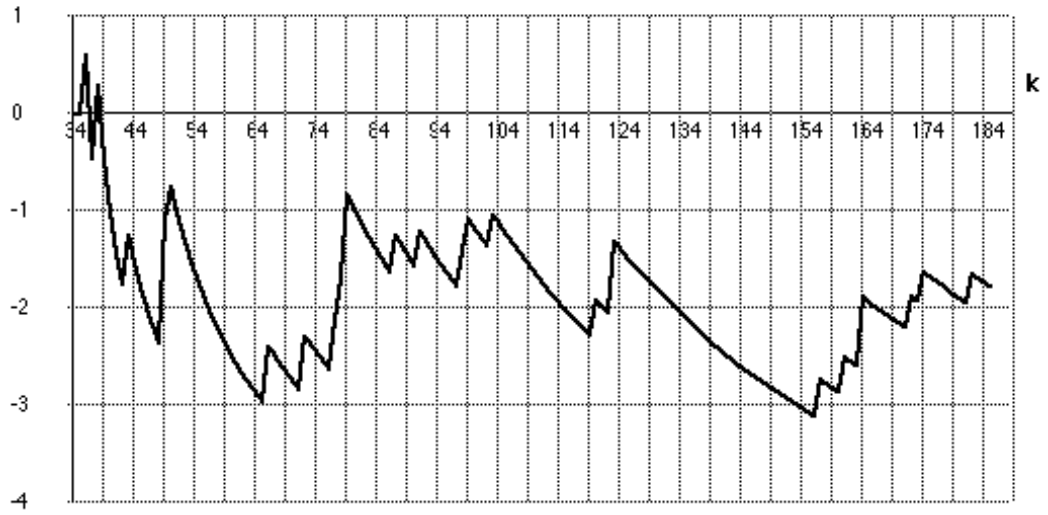


Figure 5 The Laplace test on the resticted set of failure data

On closer inspection, however, we noticed anomalous behaviour in the first two months: just two failures in the first month, and a set of eleven failures all concentrated in the second month. Thus, we decided to filter the data, by discarding the first month of operation. After this filtering, again applying the Laplace test, the

failure data relative to the last five months (33 failures) exhibited an increase in reliability, although this remained very small in the whole as shown in Figure 5.

7.5.3 Model Fitting

Subsequently as described in Section 7.4.4.2, we had to decide which model was the best one for reliability prediction in our particular context. Since there is no software reliability model that performs better than any other one in every case e we chose to run all the available models using the tool CASRE. In this case we were dealt with failure per time period data, therefore the available software reliability models are: Brooks/Motley (BM), Schneidewind (SM), NHPP (also known as Goel-Okumoto), Generalized Poisson (PM), Yamada S-Shaped (YM).

	BM bin	**BM pois**	**NHPP**	**YM**	**SM**
-ln PL	36,677 (3)	45,998 (5)	36,677 (1)	42,494 (4)	36,677 (1)
Chi-Square	4,913 (3)	4,936 (4)	4,772 (1)	26,050 (5)	4,772 (1)

Table 2 Accuracy results

As a consequence, for each model the tool automatically computes the Chi-Square test and the prequential likelihood function (specifically the -ln(PL)), (Section 7.4.4.2). We show in Table 2 the results of the accuracy analysis obtained.

We can observe that the evaluation of the Chi-Square test (in this case with 4 degrees of freedom) brings us to reject the Yamada S-Shaped model (the models fit well for the analysed data set if the Chi-Square value is within the interval [0.711, 9.49]).

Considering the value computed for (-ln PL), we can see that the NHPP and SM models gave the most accurate results for the data source. Indeed, for our data set the observation period had the same length, so NHPP and SM were equivalent [LY96 Chapter 3].

Finally, in Figure 6 we show the cumulative number of failures predicted versus the raw data and Figure 7 shows the relative error, computed as:

Eq. (13) $$\frac{(PredictedNoFailures - ActualNoFailures)}{ActualNoFailures}$$

From this experience some remarks from the comparison between SRET and the conventional approach are possible, regarding in particular the effort required to apply the SRET approach:

- SRET requires extra effort with respect to the Function Test standard process, in particular for the definition of the operational profile. If new systems have to be modelled, it will be quite difficult to find usage data to assign the right probabilities.
- The definition of test instructions can be more difficult because in SRET they have to be specified in a more abstract way (for the need to consider random variables);

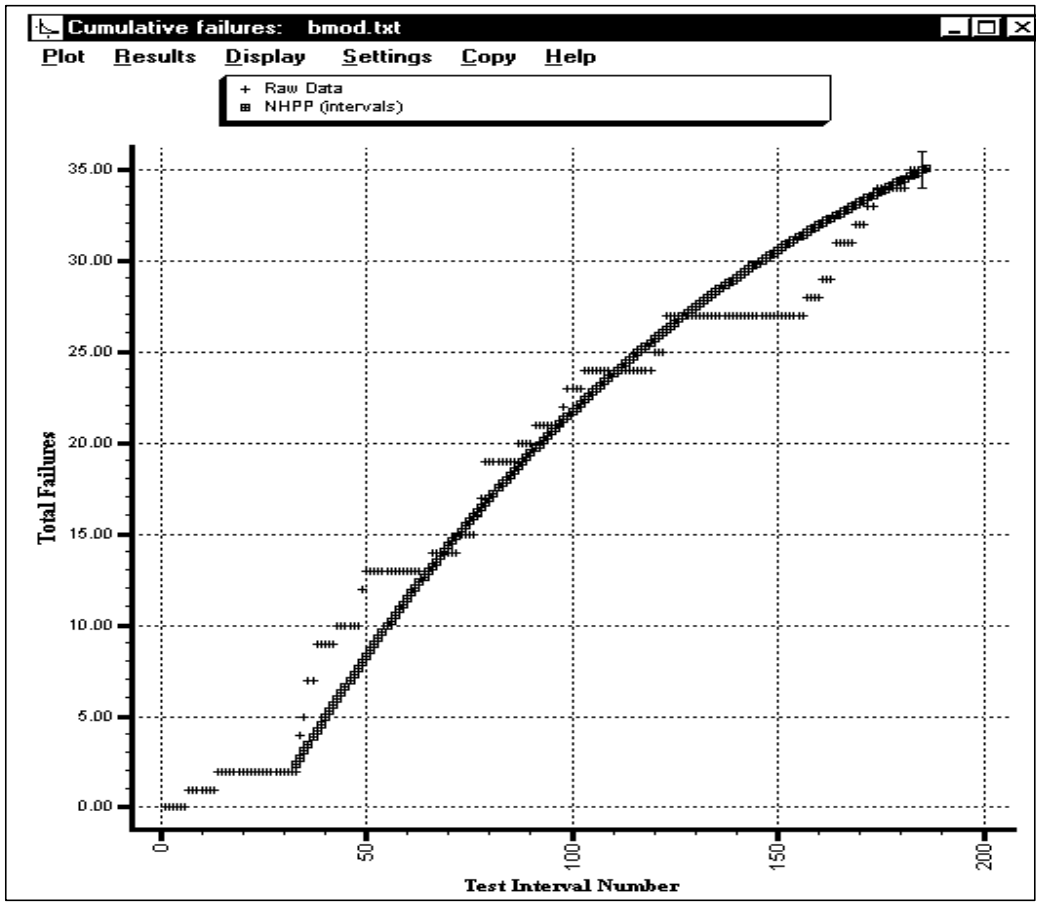


Figure 6 The cumulative number of failures predicted versus the raw data

- Considering the high number of test cases to be executed, a completely automated environment is required. The Ericsson AXE10 target environment is not completely automatable, so we had to limit the application of this case study

to the simulated environment, that is the administrative part. It was not possible to execute some categories of test cases that require to be run on the target system, such as tests for evaluating performance or involving traffic.

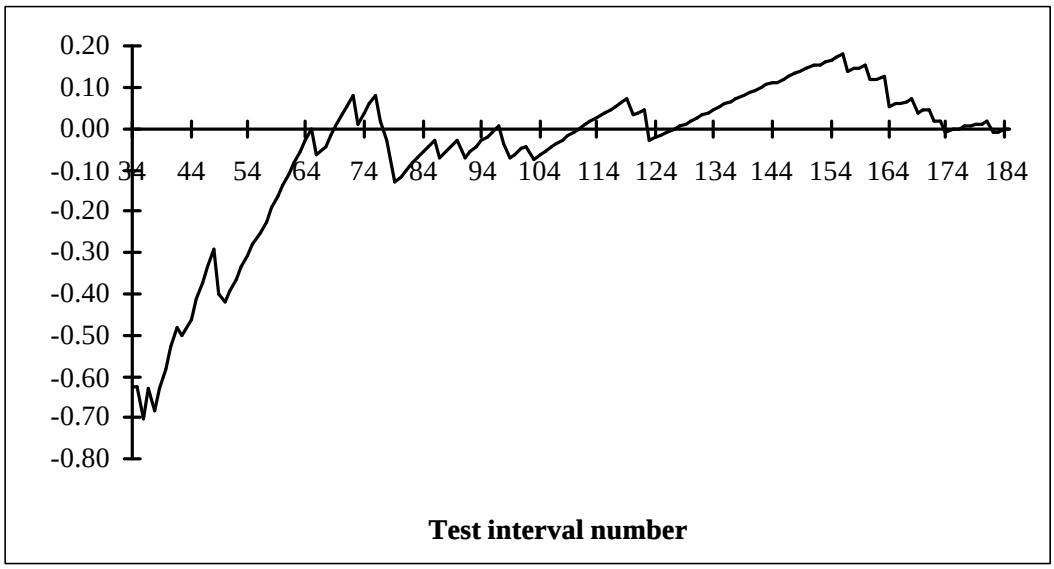


Figure 7 The relative error

Concerning the beneficial aspects, the use of the SRET approach provides:

- a better understanding of the function/feature in the early phase of the development process;
- a structured approach for identifying the test cases (we used a behavioural tree) that made the identification of test cases easier.
- a prioritisation of functions and test cases to shorten the lead time in date-driven projects. In fact, the exit criterion of the standard Function Test process requires that the entire set of test cases must be executed successfully. This criterion involves an execution time related to the number of test cases. The introduction of a priority criterion and a classification of incident severity levels allows a tester to identify exit criteria to shorten the lead time in date-driven projects. This priority criterion may be identified by using the operational profile, i.e.: the most critical set of test cases as experienced by the customers has to be executed and passed without any failures of major incident severity levels;
- an improved work organisation between developers and testers, both involved in the derivation of the operational profile. The close collaboration between testers, system engineers and product users has produced valuable side benefits, such as a

deeper understanding of user needs, less ambiguity in the specification of system requirements, and the possibility for testers to contribute to system reviews;

- expertise has been gained during this case study both with respect to the construction of an operational model, and with the use and tuning of reliability models, thus enhancing ERI's staff awareness of reliability issues, and allowing a reuse of those competencies in future projects. Should the new techniques prove successful, applying the new test technique will allow ERI to control the reliability of its software products and the associated test costs.

Summary

In this Chapter we presented the main concepts of Software Reliability Engineering focusing in particular on the SRET approach. We discussed the procedural steps necessary for defining a suitable test environment, in which operational test can be performed and reliability estimation achieved. For this we provided some basic definition of the reliability theory and an overview of the main Reliability Growth models.

We have pointed out the advantages and the difficulties in applying these models for reliability prediction, highlighting the important role of the available tools in facilitating the reliability growth model usage. For this we have discussed a procedure which depicting the steps necessary for the integrated use of two available tools, SoRel and CASRE, for obtaining the required prediction. Finally, once we applied the SRET approach to a real case study, we used the described procedure with the failure data obtained, for choosing the best reliability growth model for reliability prediction.

PART 5:
POSSIBLE IMPROVEMENTS

8 Conclusions and Future Work

Preface

This is the concluding Chapter of this Thesis, in which we present the conclusions and an ongoing experience in defining a general framework, called ``UML Combination'', for enabling the validation of component-based (CB) systems by testing them against the corresponding UML architectural specifications (Section 8.2). In particular we will readapt two previously developed tools, (Cow_Suite presented in Chapter 5) and CDT [BP03], which permits the codification and execution of test cases within a CB development process.

8.1 Conclusions

In this Thesis we have presented our journey through the world of Software Testing, ranging over many fields from definition to organization, from its applicability to analysis of its effectiveness. Adopting as a roadmap the testing phases subdivision of [BE01], we began at the planning activity and we proceeded systematically presenting new methods, approaches and tools useful to the reader for managing, controlling and evaluating Software Testing development.

These were the result of a strict collaboration with software developers looking for solutions for their problems and improvements in the different activities of the testing process. In order to respond to these needs we made a deep analysis of literature, which provided us with hints and ideas either for the definition of new methods and approaches, or for readapting and modifying already existing proposals.

The collaboration with industries imposed us two important constraints, which must be always respected even at the detriment of the quality of the possible results. These are usability, i.e. the methodologies as far as possible must adapt themselves to the modelling notations and procedures commonly used by industries and real environments and not vice versa, and automation, i.e., increasing as much as possible the mechanization in test cases derivation, execution and validation, consequently reducing the manual labour.

Taking into consideration these constraints, for our proposals we adopted the leading principle of providing readers with some easy-to-apply and low-cost methodologies, which maximize the automation and minimize as much as possible the required additional formalism or ad-hoc effort specifically for testing purposes.

Moreover for completeness' sake for each topic treated we have provided both a detailed survey of the literature useful for knowing the state of the art and for comparing our solutions, focused on putting theory into practice, with those provided by the research world, and the evaluation of the methods proposed by means of case studies also taken from a real industrial context.

In the next sections we present a summary of the proposals of this thesis with their limitations and the future work.

8.1.1 Proposals and Future Work

In this section we briefly resume the proposals presented in this Thesis, also highlighting their general limitations starting, from the test planning to the evaluation of test results.

- Test planning: we provide an original method, called Propean, based on the techniques of computer software performance engineering and queueing networks for scheduling the testing activities and distributing personnel and resources among them by considering a multiproject environment. This approach requires users confident with RT-UML for modelling the flow of activities to be performed during development and the tasks to distribute among personnel. In particular for increasing the accuracy of the prediction also associating to each activity the proper estimation, a data-base containing information derived from similar projects is necessary.
- Test Case generation: we provide a tool for the selection of functionalities to be tested and the generation of test cases, which supports the user both in the choice of the most important software elements on which the testing effort must be concentrated, and in the automatic generation of the appropriate test cases by using the available UML product specification. Thus the quality and the effectiveness of the generated tests depend strictly on the quality of the available design: in presence of an incompleteness the approach can only highlight the design deficiencies but not overcome them producing meaningful test cases. In the Thesis we describe the current status of our approach but several improvements are possible such as: implementing other strategies for test cases

selection which take also in consideration the diverse cost of each test case; providing hints or defining a common strategy for assigning the weights to the nodes; implementing methods diverse from UIT for test cases and procedures derivation. Notwithstanding the encouraging results obtained this approach needs further validations with more complex case studies

- Test results analysis: We consider non-operational and operational testing. In the former case we propose two dynamic methodologies, the One-Step and the Two-Steps Method for deriving the number of failures experienced up to the end of testing phase, by using data collected during the testing itself. In the latter the integrated application of different tools for reliability growth models selection and usage. Even if both approaches are quite general, it is worth noting that the methods for the non-operational testing have been developed in strict relation with the industrial partner which provided us the stimulus for model formulation and development. Thus the procedures adopted follows the process for collecting data of ERI which however for its simplicity can be considered quite general and representative of industrial practice.

Of course, our work is not yet concluded; many other problems remain unsolved and improvements in our methodologies are possible. The research area of Software Testing is so vast and involves so many problems that they cannot of course be exhausted in this Thesis. We have provided here our "little" contribution on some of the salient points that arose during the cooperation with software developers.

In the future, encouraged by the positive results obtained, we wish to quantitatively evaluate the proposed methodologies and approaches with further industrial case studies, and unify all the proposals of this Thesis in a unique control process useful for managing the testing phase during the entire software life cycle.

In particular in the next section we present one of the ongoing work which has the purpose of improving the strategy of test case generation in component-based environment

8.2 An ongoing Experience: UML Combination

Component-based (CB) development is one of the focal trends in software production today. Although in recent years it is attracting much interest from both academy and industry, as testified by the spreading of related events (e.g., [ICSE03], [ECBS02]), journal articles (e.g., [IEEE99], [JSS03]), and by the market launch of component-oriented technological products and platforms (e.g., CCM [COR], EJB

[EJB], COM+/.Net [NET]), research in this area is far from complete. Many topics, such as component specification, development tools, or performance predictability, are in fact still open.

In this research sphere, UML application for the specification of CB systems is just beginning. Although UML was not conceived with a CB paradigm in mind, it is very flexible and provides suitable mechanisms for extensions. To this purpose one of the emerging references is the methodology proposed by Cheesman and Daniels [CD00], called the ``UML Components", described briefly in Appendix C, which focuses both on the representation of the components and on the development process applicable for this purpose. Thus the purpose of our ongoing research is to apply this new methodology for component testing, which needs a re-evaluation to address the peculiar characteristics of CB development as for the other development phases [BP02].

In our opinion an important requirement is that the customer, on the basis of what he/she expects from a researched component or architecture, and with reference to the system specification/architecture, develops test suites easily (re)executable to evaluate the potential candidates. To facilitate the customer in this task we are implementing a general framework, called "*UML Combination*" (UML COMponent-Based INtegrAted Testing envIrONment). The purpose is the validation of component-based systems by testing them against the system's architectural specifications. In particular, starting from the original idea presented for the first time in [BMP03], we are defining the UML Combination test environment, which can be used by the software developer both for: (i) deriving the test cases and (ii) codifying and executing them.

The methodology will be the result of the combination, with the necessary adaptations, of two tools developed in previous projects: the Cow_Suite, presented in Chapter 5, which will analyze the UML components specification for selecting and generating test cases, and the CDT ([BP03] [BP02a]) which codifies the test cases and (re)executes them every time a component instance is plugged into the system.

Therefore the main requirement for applying the UML Combination will be that, by using the UML Components guidelines, the system developer specifies the design architecture of the system with particular attention to the interfaces. Then by applying UML Combination, he/she can automatically derive a meaningful set of test cases and execute them when necessary.

While the principles of integration have been settled, the combination of Cow_Suite and CDT is currently under implementation So far, the UML Combination has only been conceived and partially developed for testing a single "virtual" component in isolation, i.e. the component specified at requirement level and implemented successively by the integration of one or more real components. In Section 8.2.1 we present the current status of the implementation. We are still working on the extension of UML Combination to the test of a "subsystem", i.e. an integrated set of virtual components.

8.2.1 Proposed Approach

In this section we show briefly how we are going to combine the two components of CDT and Cow_Suite, in order to obtain the UML Combination integrated testing environment as schematized in Figure 1.

We discriminate in particular between two different levels of testing as will be described in the following sections: the test of the single virtual component, which may be obtained by one or more real components, and the test of a group of integrated virtual components (a subsystem) in the final application environment [BMP03].

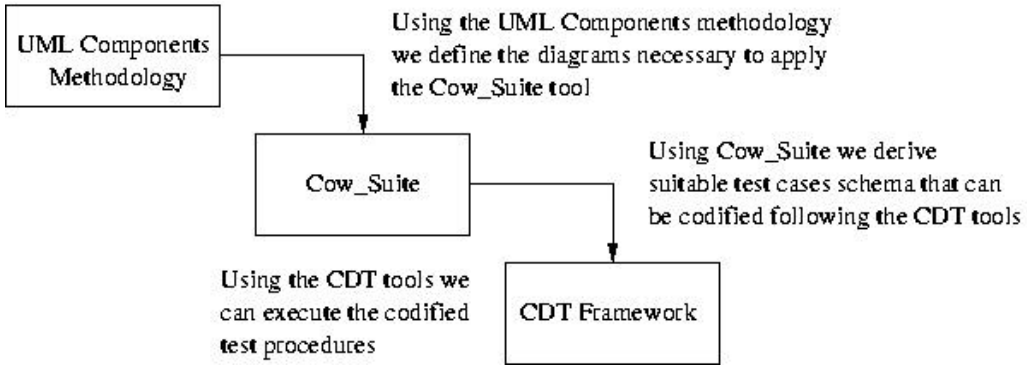


Figure 1 The overview of UML Combination.

8.2.1.1 Test of the Single Virtual Component

At this stage each component is tested singly, by means of suitable stubs when necessary. For each component we use its UML specification to derive, by the help of Cow_Suite as described in Chapter 5, the set of test procedures that will be used to verify the conformance of its instance. For this we intend to modify Cow_Suite both in the final weights distribution and in the procedure adopted for analyze every

Collaboration Diagram (CD) or Sequence Diagram (SD) in which an interface, belonging to the tested component, is involved.

Resuming what was explained in Chapter 5, starting from the main Use Case Diagram onwards, Cow_Suite uses the developed UML diagrams, and the mutual relationships within them, for organizing them into several oriented graphs. These are then explored for producing the basic hierarchical structures (trees) of the Cow_Suite approach. Every node is then annotated with its final weight representing the importance of the node itself, belonging to the [0,1] interval. In the CB environment, in the case of a single virtual component, the role of the final weights will be improved with respect to the original release of the Cow Suite tool. They in fact will be used for associating an importance factor to the methods of its interfaces, as described below, and for distributing the test cases accordingly:

1. For each CD in which an interface of the specified component appears, the final weight of the CD is distributed among the invoked methods belonging to the considered interface;
2. For each method the sum of all the values obtained in the previous step is derived;
3. For each interface the sum of the values associated to its methods is normalized to 1

The obtained values will then be used for applying the two different test strategies supplied by the Cow_Suite tool: a fixed number of test cases to be executed or a fixed percentage of methods to coverage. As described in detail in Chapter 5 in the former the tool will select the most suitable distribution of the test cases among the methods on the basis of their weights; in the latter it will highlight the most critical methods the proper test cases distribution.

For automatically constructing the tests in both cases we also decided to implement within the Cow_Suite tool the original version of the UIT methodology as presented in [BB00] and summarized in Appendix B. Specifically, we first isolate the CDs/SDs in which an interface belonging to the tested component is involved and then by applying the original UIT, we derive from them the ordering of messages and the feasibility conditions used in the generation of test cases. The choices, useful for the definition of the test procedures, will be derived either as usual by the interaction with the user or by the analysis of the contract associated to each method [BLS02]. To this purpose particular attention is dedicated to the preconditions, which can specify parameter intervals or values useful for test cases generation.

Finally the test procedures will be codified using the CDT framework, without needing to refer to any particular real implementation. In this framework the invocations of the test cases refer to the interfaces of the virtual component, which will be implemented using yet-unknown components. It is important to note that by assembling prefabricated components to form a virtual component, it is likely that the real implementation could supply more functionalities than those required. In this case the established set of test cases will only stress the functionalities defined in the UML specification.

8.2.1.2 Test of a Group of Integrated Virtual Components

We are still working on testing a group of integrated components, because it brings up major problems to maintain the original philosophy of the Cow_Suite (Chapter 5). The main difficulty we are facing is the inability to define meaningful test cases when there are exceptional conditions in the UCs definition. In this context during the test cases run, it is not possible to control the path (i.e. the sequence of invocations) that the test will follow inside the composed system.

Thus we are attempting to find solutions to this inconsiderable problem, which is intrinsically related to the knowledge of the states in the black box component. We present here two possible approaches which we are studying.

The first is to review the role of Cow_Suite in the integration phase, and assuming the weights associated with the CDs/SDs as a sort of “critical profile” indicating the importance of a particular scenario in the system. In this case the test phase is halted only when each CD/SD has been covered for the specified number of times but this solution is clearly in contrast with the original goal of Cow_Suite.

A second approach under evaluation is to modify the hypothesis concerning the component model. So far we have always supposed that a real component is only constituted by two “sets” of signatures, respectively representing the provided and the required services, and by a brief textual description of the functions performed. In order to address the “control path problem”, the solution analyzed is to require that a real component implements particular interfaces (probes) that permit investigating the state of an instance of the components, and foresees a sort of parameterization in the definition of the test cases.

In both cases, it can be useful for the tester team to recover the test cases developed during the verification of the single components in order to obtain parameters for the test procedures generation.

Moreover, the diagrams used to derive test procedures can be used fruitfully, also as a guideline for the integration workflow. In this manner we obtain a functionally driven workflow, rather than a structural one (as for instance it would be if class diagram were used).

Appendix A. An overview of EG and QN

In this appendix we provide a brief overview of execution graph and queueing network modelling referring the reader for more details [SM90, LZS84].

The Execution Graph (EG) represents the software execution model and provides a graphical representation of the processing steps. Like the UML activity diagram (Chapter 3) it consists of a set of nodes, representing the software workload components, and a set of arcs, representing the transfer of control.

The software workload components, which can be single instructions or entire procedures, depending on the granularity adopted for the model [SM90], allow modelling software at different levels of detail. In particular EGs include several types of nodes (or blocks), such as basic, cycle, conditional, fork and join nodes, described briefly in Figure 1.

There are two restrictions on the construction of the EG which have the purpose of simplifying the solution algorithms [SM90]:

1. *Initial node restriction:* there is only one initial node representing the first processing step executed in the graph.
2. *Loop restriction:* all loops in the graph must be repetition loops

It is worth noting that the execution graph only models those paths fundamental for performance and not all the possible paths, and degree of detail to represent in the graph is left to the user. Consequently different execution graphs can model the same software (there is not a unique representation).

Considering instead the Queueing network modelling the systems are defined by using Queuing Networks (QNs), which are a collection of interconnected nodes representing the service centres, i.e., the system resources, and customers i.e., the users or transactions. The nodes (the service centre) can be [LZS84]:

Single Service Centre: Customers arrive at the service centre, possibly wait in the queue, receive service from the server, and depart. This model has two parameters: *workload intensity*, i.e. the rate at which customers arrive, and the *service demand*, i.e., the average service requirement of a customer. By solving this

model the performance measures obtained are: *utilization*, the proportion of time the server is busy; *residence time*, the average time spent at the service centre by a customer; *queue length*, the average number of customers at the service centre; *throughput*, the rate at which customers pass through the service centre

Multiple Service Centres: The parameters of this model are analogous to those of the previous but in this case the *service demand*, requires separate values for each service centre.

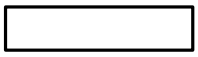
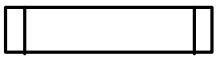
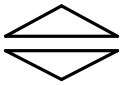
Name	Symbol	Meaning
Basic node		The processing step at lowest level of detail
Expanded Node		The processing step has been refined and detailed in a sub-EG
Repetition Node		The following nodes are repeated n times.
Case Node		Every node has specific execution probability, and is executed according with associated condition
State Node		lock-free, fork-join, send-receive, acquire-release
Split Node		Attached nodes are new processing threads. They need not all complete before proceeding
Arc		The execution goes from the arc to the destination node
Call-return arc		When the operations are completed the execution from the destination come back to the origin
Driver acr		When the operations are completed the execution from the destination come back to the node X
Dummy		There is not processing time associated to the arc

Figure 1 The elements of an Execution Graph

Generally two categories of QN can be distinguished depending on the type of transactions (users) considered. The first is the *open system,* in which all the users can leave the system; the second is the *closed system* in which no user can leave the system. In this case the number of users is fixed.

In each of them a node may be a *Non-Blocking* (or infinite capacity) node, i.e. it can accommodate any number of users waiting to be served. When the storage space in front of the server is finite a node is called a *Blocking* (finite capacities) node, i.e., a prefixed limit is imposed on the number of users waiting to receive a service. Considering the closed QN if a node can hold all the customers in the network, it can be defined as an infinite capacity node. QN with finite capacities are used for representing more realistic models of flow systems. Blocking arises because the flow of users through a queue may be halted if the destination queue has reached its capacity.

Several types of blocking may occur: *Transfer Blocking,* the customer after getting service at the source node waits, blocking the server, until there is room in the destination queue; *Repetitive Service,* the blocked customer proceeds to receive another service at the source node itself; *Rejection Blocking,* the customer attempting to enter a full queue is lost.

QN models of systems are useful as an analytical or a simulation based analysis of their performance. In particular depending on the types of the component queues in the network, different analytical algorithms may be used to obtain exact or approximate results both for the performance of the individual queues and for the overall system.

Appendix B. The UIT Methodology

The Use Interaction Test methodology, (UIT), presented for the first time in [BB00] is based exclusively on the analysis of SDs from which the relevant information to automatically construct Test Cases and subsequently Test Procedures are derived. Each SD describes a particular system scenario and explains how a Use Case is realized by the interactions of objects and actors. The objects involved in a SD are the components that provide for and execute the functionality described in the UC, through elaborations and message exchanges; therefore they are precisely the elements to be tested.

Thus the core of UIT methodology is the analysis of SDs as set forth in [JGP98], whose authors suggest how to define tests by considering the different Messages_Sequences from a possible input state, or from a system input sent by an actor. We report in the following the stepwise methodology applied to the SDs for the derivation first of the Test Cases and then of Test Procedures:

1. **Find out *Test Units.*** Observing a SD along its horizontal axis, see Figure 1 arrow (a), we can identify a set *Test Units*. Each object, which interacts through messages with other objects, represents an item that can be separately tested in order to examine a possible use of the system, and it is identified as a Test Unit.
2. **Identify *Interactions Categories*.** All messages entering in the selected Test Units are called *Interactions Categories*. In fact, as described in [UML], a message is a communication where the sender object invokes an action, a service belonging to the receiver object (our Test Unit), which will perform it. Knowing the set of Interactions Categories means catching, and thus testing, all the possible interactions among the Test Units under analysis and the other objects. For example, if we consider the Test Unit `LogonForm` in Figure 1, the observable Interactions Categories are:
   - open()
   - enterUserName (uid)
   - enterPassword (pwd)

- loginUser()
- validateuserIDPassword(uid, pwd)
- setupSecurityContext()

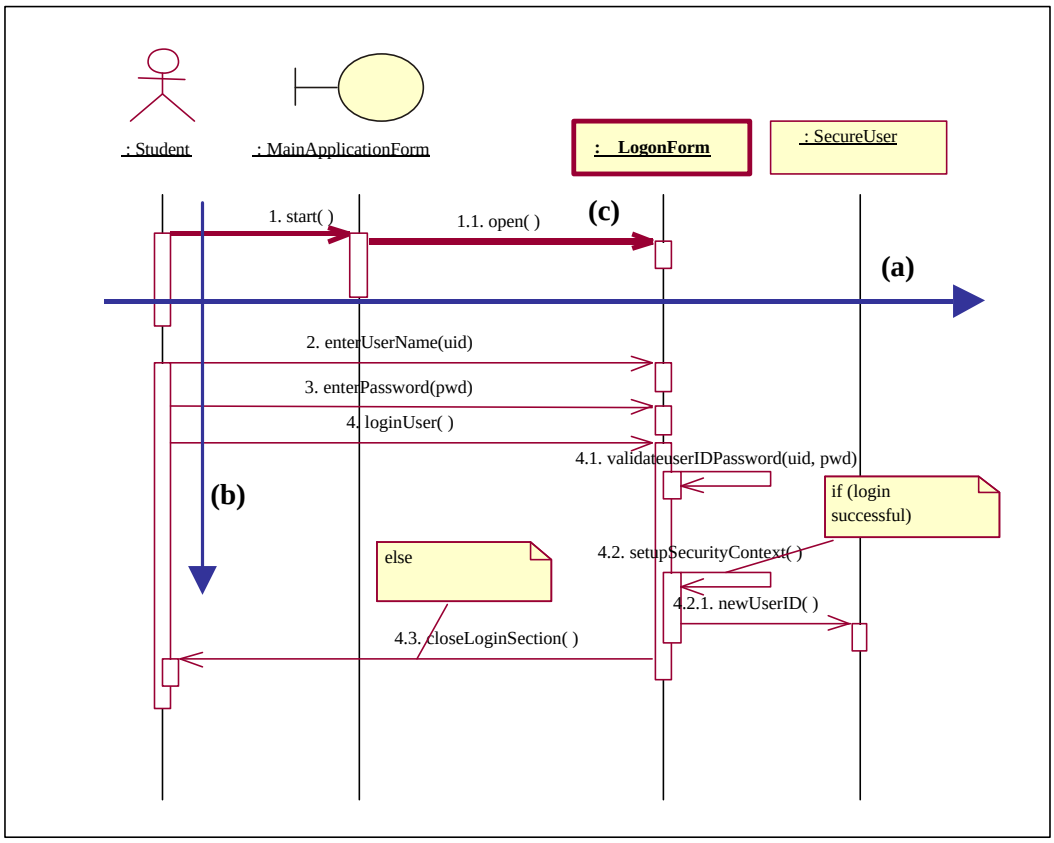


Figure 1 Sequence Diagram "Login-Main Flow" from CRS example described in Section 5.7.1

3. **Identify *Settings Categories.*** Besides, for the selected Test Units, we define the Settings Categories as the values, parameters or data structures, that can influence its interactions towards others objects. They can be determined:
   - From the Interactions Categories, by considering their input parameters;
   - From the analysis of the Class Diagram (if any) to which the Test Unit belongs, by examining the attributes and data structures that can affect the observed interactions.

   In our example, the Settings Categories for the Test Unit `LogonForm` are: `uid` and `pwd`.
4. **Define *Messages_Sequences***. Observing the vertical temporal order of the messages along the studied Test Unit's lifeline, Figure 1 arrow (b), a set of

Messages_Sequences can be detected. A Messages_Sequence, is the set of messages composed of: a message entering the Test Unit, not yet involved in the construction of other Messages_Sequences, plus all the messages (if any) belonging to its activation bounded by the focus of control region. A Messages_Sequence therefore represents behaviour to be tested and describes the interactions necessary in order to realize specific system functionality between the Test Unit and the other objects. In Figure 1 the arrow (c) {`1.start(), 1.1 open()`} represents one of the possible Messages_Sequences for the `LogonForm` Test Unit.

5. **Construct *Test Cases*.** For each identified Messages_Sequence, a Test Case can be generated. It contains the list of all Settings and Interactions Categories involved in the Messages_Sequence and their values. Figure 2 shows one of the Test Cases derived from the above example.

Test Case
Description:
Precondition:
Flow of Events (Messages_Sequences):
loginUser()
validateuserIDPassword(uid, pwd)
setupSecurityContext()
newUserID()
Categories:
Settings:
uid
pwd
Interactions:
loginuser
validateuserIDPassword
setupSecurityContext
newUserID
PostCondition:
Comment:

Figure 2 An example of a Test Case for the Test Unit `LogonForm`

6. **Analyse possible subcases**. The messages involved in a Test Case may contain some feasibility conditions. These conditions are usually described in the message notes or in the message specification formally expressed using the OCL notation [WK99]. If these feasibility conditions exist, a Test Case is divided into subcases, corresponding to the different possible choice values. Referring to Figure 1, the condition value of `login successful` differentiates execution

of the Messages_Sequence starting with message 4. In this case, if the condition is *true* we have the Test Case 1.1 with the Messages_Sequence:

4.loginUser()
4.1 validateuserIDPassword(uid, pwd)
4.2 SetupSecurityContext
4.2.1 newUserID

While in the opposite case the Test Case 1.2 contains the following Messages_Sequence:

4.loginUser()
4.1 validateuserIDPassword(uid, pwd)
4.3 CloseLoginSection()

7. **Determine *Choices:*** for each Category (both Settings and Interactions) belonging to a Test Case, the possible choices are identified as follows:
   - For the Interactions Categories, they represent the list of specific situations, relevant cases in which the messages can occur;
   - For the Settings Categories, they are the set or range of input data that parameters or data structures can assume. In Figure 3 we report the choices values for the Test Unit `LogonForm.`
8. **Determine *Constraints*** among choices: the values associated to the choices of the setting and Interaction categories of a Test Case may turn out to be either contradictory or even meaningless. This can be avoided by adding feasibility conditions to the categories choices as suggested in the Category Partition methodology. These constraints are specified, by assigning *Properties* or *IF Selectors* to choices. Specifically the Properties are used for checking the compatibility of a choice with the others belonging to the same Test Case, and the IF Selectors are used to validate the conjunction of properties previously assigned to other choices, as reported in Figure 1 in the sentences in square brackets.
9. **Derive *Test Procedures***: Finally, using these choice values, a Test Procedure can be generated for each possible combination of compatible choices, for every category involved in a Test Case. Figure 4 shows one of the final resulting Test Procedures for `LogonForm`. For each analysed Test Unit, all the meaningful Test Procedures are collected in the *Test Suite*.

Test Unit *LogonForm*

Interactions Categories:

open()
- access from a Student

enterUserName()
- access request of a new user [Property new]
- access request of a registered user [Property registered]
- access request of a not allowed user [Property notAllowed]
- access request of a expired account user [Property expiredAccount]

enterPassword()
- access request with correct password [Property registered]
- access request with wrong password [Property registered]

loginUser
- access request of a new user [Property new]
- access request of a registered user [Property registered]
- access request of a not allowed user [Property notAllowed]
- access request of a expired account user [Property expiredAccount]

validateUserIDPassword
- access validation of a new user [IF new]
- access validation of a registered user (correct uid and pwd) [IF registered]
- access validation of a registered user (wrong uid or pwd) [IF registered]
- access validation of a not allowed user [IF notAllowed]
- access validation of a expired account user [IF expiredAccount]

setupSecurityContext
- successful access new user [IF new]
- successful access of a registered user [IF registered]

Settings Categories:

uid
- m.Jackson
- f_smith
- Paul_white
- S_71whatson ……..

pwd
- m565jkrm
- annamaria
- p71271
- 12.2.73 ……..

Figure 3 Choice values for Test Unit `LogonForm`

Test Procedure

loginUser()

access request of a registered user

validateuserIDPassword(uid, pwd)

access validation of a registered user (correct uid and pwd)

setupSecurityContext()

rID()

access of a new user

uid

f_smith

pwd

m56ikrm

Figure 4 Test Procedure example

Appendix C. UML Components

We report briefly the main details of methodology proposed by Cheesman and Daniels [CD00], called the UML Components, which focuses both on the representation of the components and on the process development applicable for this purpose. This is an expansion of the classical notation of UML, which includes the extensions required for specifying the components, i.e. their specification, interface, implementations and the objects component obtained.

The idea of UML components was born in the middle of the 1990 from the collaboration of many minds which focused attention both on the representation of the components by using UML and on the process development applicable for this purpose. Without aiming to present here an exhaustive survey of the literature, one of the first works in the application of object-oriented practices is that of Cook and Daniels [CD94] in which the Syntropy methodology was developed. This represents a common base for many recent developments, like Catalysis [DW99] or the UML itself, and is the direct ancestor of the Object Constraint Language (OCL) [WK99]. From Catalysis another important methodology for defining the development of the component based systems descends, called Advisor [ADV], which largely influences, together with the Rational Unified Process [RUP] the actual definition of UML components.

Cheesman and Daniels attempt to unify all this knowledge and experience in their book [CD00], presenting an easy-to-apply specification process for component-based systems, which focuses only on the modeling of the software applications, completely ignoring their final implementation.

Starting from the requirement specification, we briefly report here the main details of the specification process adopted by Cheesman and Daniels and the UML extensions required. The specification process is divided into different workflows (following the definition of the RUP: a sequence of activities that produces a result of observable value) interacting together, each one specified by using UML notation.

The tasks of the *requirement workflow* are the business concept model and the use case model. The former is a conceptual model, which specifies the key concepts, their relations and a common vocabulary useful for avoiding misunderstanding and ambiguities. It is represented by a class model, but the classes involved, as well as their associations, are only conceptual and not related to the specification.

Instead the use case model represents the interaction of the system with the external users. It is represented by a Use Case Diagram, in which each Use Case is related to a different requirement. The system behaviour and main exceptions are represented for each Use Case in the associated scenario, following the textual structure of the Cockburn's Use Cases [CO01]

The *specification workflow* is subdivided into three phases:

i. The identification of the components: starting from the requirements, an initial system architecture is produced;
ii. The interactions between the components, which identify the system operations and responsibilities;
iii. The specification of the components, which specifies the operations and interfaces of the components themselves.

A business model, represented by a class diagram, is used for modelling the business information. The classes involved are defined at the specification level, with no relation to a specific language. The notation used for the component interfaces differs from that defined in the standard UML, in which the interfaces represent implementation constructs typical of the OO languages and that do not require attributes or associations. In the UML Components, an interface specification consists of: the type, the information model (the attributes, the interface roles in the association and their types), the specification of the operation (prototypes, pre- and post-conditions), and the invariants. All this information is grouped together in a package representing an interface specification, which can also import information from other packages.

In UML Components, even the concept of a component is quite different than in the standard UML, because it is completely independent of the implementation. To differentiate the specification of a component from its implementation or the installed component, a new stereotype <<comp spec>> is introduced which has a set of interface types. The ways in which the components interact via the interfaces are finally described using collaboration or sequence diagrams.

The *provisioning workflow* is aimed at ensuring that the released software is consistent with the given specification of the components. For this purpose the components can be implemented, bought, readapted or derived from the integration of existing ones

Finally the *integration workflow* connects the various components, the user interface, the application logic and the existing software in order to obtain an efficient application.

Bibliography

[ABK94] Ammann, P., Brilliant, S.S, Knight, J.C. “The Effect of Imperfect Error Detection on Reliability Assessment via Life Testing”. IEEE Transaction On Software Engineering Vol. 20, No. 2, pp. 142-148 1994

[ABP02] Antoniol, G., Briand, L. C., Di Penta, M. , Labiche, Y. “A Case Study Using the Round-Trip Strategy for State-Based Class Testing”. IEEE Int. Symposium on Software Reliability Engineering, Anapolis, USA, 2002.

[ACL86] Abdel-Ghaly, A..A., Chan, P.Y., Littlewood, B., "Evaluation of Competing Software Reliability Predictions", IEEE Transaction on Software Engineering, Vol. 12, No. 9, September 1986.

[ACL96] Abdel-Ghaly, A., Chan, P.Y., Littlewood, B. “Evaluation of Competing Software Reliability Predictions”, IEEE Transaction On Software Engineering, Vol. SE-12, No. 9, September 1996, pp. 950-967.

[ACL01] Antoniol G., Casazza G., Di Lucca G.A., Di Penta M., Rago F. “A Queue Theory-Based Approach to Staff Software Maintenance Centers”. In Proceedings of IEEE International Conference on Software Maintenance, ICSM 2001, 6-10 November 2001, Florence, Italy.

[AD84] Adams, E.N. “Optimizing Preventive Maintenance of Software Products”, IBM Journal of Research and Development, 28 (1), pp.2-14, 1984.

[ADV] Sterling Software Component-Based Development Method. On line at http://www.ca.com/products

[AGE02] AGEDIS. Available at http://www.agedis.de/index.shtml

[AI91] ANSI/IEEE “Standard Glossary of Software Engineering Terminology” STD-729-1991. ANSI/IEEE 1991

[AL90] Allen, A.O. “Probability, Statistics, and Queuing Theory with Computer Science Applications”. Academic Press, 1990.

[AMN95] Adler, P.S., Mandelbaum, A., Nguyen, V., Schwerer, E. “From Project to Process Management: An Empirically Based Framework for Analysing Product Development Time”. Management Science, Vol. 42, 1995, pp.458-484.

[ANA] Analyst Pro - Powerful UML Tool. Available at: http://www.analysttool.com/UML_Usecase.html

[BB00] Basanieri, F., Bertolino, A. “A Practical Approach to UML-based Derivation of Integration Tests”. Proceeding of QWE2000, Bruxelles, November 20-24, 3T.

[BBM01] Basanieri, F., Bertolino, A., Marchetti, E. “CoWTeSt: A Cost Weighted Test Strategy”. Proceeding of ESCOM-SCOPE 2001, London, England, 2-4 April 2001.

[BBM02] Basanieri, F., Bertolino, A., Marchetti, “The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects", UML 2002, LNCS 2460, Dresden, Germany, September 30 - October 4, 2002, pp. 383-397.

[BBM02a] Basanieri F., Bertolino A., Marchetti, E., Mirandola, R.” Automating the Management of Teams and Tasks in Software Multiprojects using UML and Queueing Networks”. Proceedings of 3rd ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD, Madrid, Spain, June 2002

[BBM03] Basanieri F., Bertolino A., Marchetti, E., Mirandola, R. ”UML-based Performance Analysis Techniques Applied to Software Multiprojects Management” ACIS International Journal of Computer & Information Science (IJCIS), Vol.4, No.1, March 2003, pp. 1-13

[BCI00] Bertolino, A., Corradini, F., Inverardi, P., Muccini, H. “Deriving Test Plans from Architectural Descriptions”. Proceeding of ICSE 2000, Limerick, June 2000, pp 220-229.

[BCL90] Brocklehurst, S., Chan, P.Y., Littlewood, B. Snell, J. “Recalibrating Software Reliability Models” IEEE Transaction on Software Engineering, Vol. 16, No.4, April 1990, pp.458-469.

[BE90] Beizer, B. “Software Testing Techniques” 2nd Edition, International Thomson Computer Press 1990

[BE93] Berard E.V “Essays on Object-Oriented Software Engineering”, Vol. 1. Prentice Hall, 1993.

[BE01] Bertolino A., “Knowledge Area Description of Software Testing”. Chapter 5 of SWEBOK: The Guide to the Software Engineering Body of Knowledge. Joint IEEE-ACM Software Engineering Coordination Committee. (2001). On line at http://www.swebok.org/.

[BE03] Bertolino, A. "Software Testing Research and Practice", 10th International Workshop on Abstract State Machines ASM 2003, Taormina, Italy, March 3-7, 2003, LNCS 2589, p. 1-21.

[BFS90] Black, T.A., Fine, C.H., Sachs, E.M. “A Method for Systems Design Using Precedence Relationships: An Application to Automotive Brake Systems”. M.I.T. Sloan School of Management, Cambridge, MA, Working Paper no. 3208, 1990.

[BI99] Binder, R. V. “Testing Object-Oriented Systems - Models, Patterns, and Tools”, Addison-Wesley, 1999.

[BIL03] Basanieri F., Iani P., Lombardi G., Marchetti E. “An industrial experience in comparing manual vs. automatic test cases generation” Proceedings of 4th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD, Lubeck, Germany October 2003.

[BIM01] Bertolino, A., Inverardi, P., Muccini ”An Explorative Journey from Architectural Test Definition Down to Code Tests Execution”. Proceeding of ICSE 2001, Toronto, Canada, 12-19 May, pp 211-220.

[BL92] Brocklehurst, S., Littlewood, B. "New Ways to Get Accurate Reliability Measures", IEEE Software, Vol. 9, No. 4, July 1992, pp. 34-42.

[BL01] Briand, L.C, Labiche, Y. “A UML-Based Approach to System Testing”. Journal of Software and Systems Modeling (SoSyM) Vol. 1 No.1 2002 pp. 10-42.

[BLA98] Barghout, M. Littlewood, B., Abdel-Ghaly, A. A "Non-Parametric Order Statistics Software Reliability Model" Software Testing, Verification & Reliability 8(3): 113-132 1998

[BLM98] Bertolino A., Lombardi G., Marchetti E., Peciola, E. "Introducing a Reliability Measurement Program into an Industrial Context", Proc. of ESCOM-ENCRESS 98, Roma, May 27-29, pp. 277-286.

[BLM02] Bertolino A., Lombardi G., Marchetti E., Mirandola R. "Performance Analysis of the Rational Unified<<Process Product>>", In Proceedings of 12-th International Workshop on Software Measurement, IWSM 2002, Magdeburg, Germany, October 2002

[BLP01] Bori S., Lores J. Pascual R. Roures E. "PROMAN, Planning Production Management and Control System with Intelligent Interface and Advanced Forecast". In Proceedings of ETFA 2001, 8-th IEEE International Conference on Emerging technologies and Factory Automation, Antibes (France), 15-18 October 2001.

[BLS02] Briand, L.C., Labiche, Y., Sun, H. "Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code", Proc. of ISSTA 2002, Roma, Italy, July 22-24, 2002, pp. 70-80.

[BM80] Brooks, W.D., Motley R.W. " Analysis of Discrete Software Reliability Models" Rome Air Development Center Technical Report, RADC-TR-80-84, April 1980

[BM98] Bertolino, A., Marchetti, E. "A Simple Model to Predict How Many more Failures Will Appear in Testing", Proc. Quality Week Europe '98, Brussels 9-13 November 1998, paper 9T.

[BMB96] Briand, L.C., Morasca, S., Basili, V.R., "Property-Based Software Engineering Measurement", IEEE Transactions on Software Engineering, Vol. 22, No 1, January 1996, pp. 68-86

[BMM02] Bertolino A., Marchetti, E., Mirandola, R. "Real-Time UML-based Performance Engineering to Aid Manager's Decisions in Multi-project Planning" in Proceedings of Third ACM International Workshop on Software and Performance, WOSP 2002, Rome, Italy, July, 2002

[BMM02a] Bertolino, A., Marchetti, E., Mirandola, R., Lombardi, G., Peciola, E. "Experience of Applying Statistical Control Techniques to the Function Test Phase of a Large Telecommunications System", IEE Proc. Software, Vol. 149, No. 4, August, 2002, p. 93-101

[BMM03] Bertolino, A., Marchetti, E., Mirandola, R. "Propean, a RT-UML based Approach to Help Manager's Decision-making". Technical Report ISTI-2003-TR-19 2003

[BMP03] Bertolino, A., Marchetti, E., Polini, A. "Integration of "Components" to Test Software Components"', in Proceedings of TACoS 2003, Workshop at ETAPS 2003, Warsaw, Poland, April 13th, 2003, pp. 51-61.

[BO88] Boehm, B.W. "A spiral model of Software Development and Enhancement". IEEE Computer, May 1988.

[BO96] Burr, A. Owen, M. "Statistical Method for Software Quality: Using Metrics for Process Improvement". Int. Thomson Computer Press, 1996.

[BP02] Bertolino, A., Polini, A, "Re-thinking the Development Process of Component-Based Software" in Proceeding of ECBS April 10-11, 2002, Lund, Sweden

[BP02a] Bertolino, A., Polini, A., "WCT: a Wrapper for Component Testing", Proceedings of Fidji'2002 in LNCS 2604, Luxembourg, November 28-29, 2002, pp. 165-174.

[BP03] Bertolino, A., Polini, A., “A Framework for Component Deployment Testing”. In Proceeding of ICSE2003, Portland, USA, May 3-10, 2003.

[BR01] Browning T. Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Direction. IEEE Transaction on Engineering Management (Vol. 48, 2001), 292-306.

[BRJ98] Booch, G., Rumbaugh, J., Jacobson, I., “UML User Guide”. Addison-Wesley Longman, 1998.

[BS96] Bertolino, A.. Strigini, L. "Predicting Software Reliability from Testing Taking into Account Other Knowledge about a Program”. 9th International Software Quality Week, San Francisco, May 21-24, 1996, paper 5T1

[CA98] Cai K. Y. “On Estimating the Number of Defects Remaining in the Software”, J. System Software, Vol. 40, No. 2, pp. 93-114, February 1998.

[CAG99] Crétois, E., El Aroui, M.A., Gaudoin O.: "Software reliability model selection: a new look on the U-plot method" 5th ISSAT International Conference on Reliability and Quality in Design, Las Vegas, USA, August 1999.

[CAS00] CASRE: Computer Aided Software Reliability Estimation; Available at http://www.openchannelfoundation.org/projects/CASRE_3.0/. Copyright 2000

[CCT02] Chan W.K., Chen T.Y. Tse T. H. “An Overview of Integration Testing Techniques for Object-Oriented Programs” Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science (ICIS 2002), Mt. Pleasant, Michigan, 2002

[CD94] Cook S., Daniels J. “Designing Object Systems”. Prentice Hall, 1994

[CD00] Cheesman, J., Daniels, J., “UML Components - a Simple Process for Specifying Component-Based Software”, Addison-Wesley, 2000.

[CDM02] Cangussu, J.W., DeCarlo, R.A., Mathur, A.P. “A Formal Model of the Software Test Process” IEEE Transactions on Software Engineering, Vol. 28, No. 8, August 2002

[CDS86] Conte, S.D., Dunsmore, H.E, Shen,V.Y. “Software Engineering Metrics and Models”, The Benjamin/Cummings Publishing Co., Menlo Park, Ca, 1986.

[CLL02] Choi, J.D., Lee, K. Loginov, A., O’Callahan, R., Sarkar, V. Sridharan M “Analysis of object-oriented programs: efficient and precise datarace detection for multithreaded object oriented programs”. In Proceeding of the ACM SIGPLAN 2002 New York, 2002.pp. 258–269.

[CLR01] Cormen, T.H., Leiserson, C.E., Rivest R.L., Stein, C. “Introduction to Algorithms”, Second Edition, The MIT Press and McGraw-Hill, 2001.

[CM02] Cortellessa, V., Mirandola, R. “PRIMA-UML: a Performance Validation Incremental Methodology on Early UML Diagrams”. Science of Computer Programming, 44 (2002), 101-129, July 2002, Elsevier Science

[CO01] Cockburn, A., “Writing Effective Use Cases”, Addison-Wesley, 2001

[COR] CORBA Component Model specifications On line at: http://www.omg.org/technology/documents/formal/compon\-ents.htm

[CRS] Course Registration System for Wylie College. On-line at http://www.rational.com/products/rup/resourse_center/examples.jsp.

[CT98] Carver R. H. Tai. K.C.”Use of sequencing constraints for specification-based testing of concurrent programs”. IEEE Transactions on Software Engineering, Vol. 24 No.6 pp.471–490, 1998.

[CT01] Chevalley. P., Thévenod-Fosse, P. “Automated generation of statistical test cases from UML state diagrams” 25th Annual International Computer Software & Applications Conference (COMPSAC'01), Chicago (USA), 8-12 October 2001, pp.205-214

[CY96] Chen, T.Y., Yu, Y.T.: ”On the Expected Number of Failures Detected by Subdomain Testing and Random Testing”. IEEE Trans. Software Engineering. Vol. 22, No. 2 (1996) 109-119.

[DE85] Dean, B.V. “Project Management: Methods and Studies”. North-Holland, Amsterdam 1985.

[DI70] Dijkstra, E.W. “Notes on Structured Programming” T.H. Rep. 70-WSK03 1970. Available at http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[DO99] Dorling A. et al., ‘SPICE, the Theory & Practice of Software Process Improvement”. IEEE Computer Society. 1999.

[DRW02] Dunsmore, A., Roper, M., Wood, M. “Further Investigation into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection”. Proceedings of 24th International Conference on Software Engineering, Orlando, FL, USA 2002 pp. 47-57

[DTG01] Dickinson M., Thornton A.C., Graves S. “Technology Portfolio Management: Optimizing Interdependent Projects over Multiple Time Periods”. IEEE Transaction on Engineering Management (Vol. 48, 2001), 518-527.

[DU64] Duane, J.T. “Learning Cure Approach to Reliability Monitoring”. IEEE Transaction on Aerospace, Vol. 2, 1964, pp. 563-566

[DW99] D’Souza, D. Willis A.C. “Objects, Components, and Frameworks with UML: The Catalysis℠ Approach” Addison Wesley, 1999

[ECBS02] ECBS 2002 “Workshop on CBSE: Composing Systems from Components”, April 10-11, 2002, Lund, Sweden.

[EJB] EJB: Enterprise Java Bean Technology. Available at http://java.sun.com/products/ejb/

[ELM01] El-Emam K.: “Knowledge Area Description of Software Engineering Process“ Chapter 9 of SWEBOK: The Guide to the Software Engineering Body of Knowledge. Joint IEEE-ACM Software Engineering Coordination Committee. (2001). On line at http://www.swebok.org/.

[EP00] Eriksson H.E., Penker M. “Business Modeling with UML”, Wiley Comp. Pub., (2000).

[EW03] Evans, I., Warden, R. “Focus on UML Testing Strategies” UNICOM The Tester’s Bulletin, Seventh Issue - February 2003.

[FBK91] Fujiwara, S., Bochmann, G. v., Khendek, F., Amalou, M. and Ghedamsi, A., Test selection based on finite state models, IEEE Transactions on Software Engineering, Vol.17, no.6, June 1991, pp. 591-603.

[FHL98] Frankl, P.G, Hamlet, R.G., Littlewood, B., Strigini, L. "Evaluating Testing Methods by Delivered Reliability," IEEE Transactions on Software Engineering (ICSE '97), V. 24, 8, Aug. 1998, pp. 586-602,

[FL02] Fraikin, F., Leonhardt, T. “SeDiTeC - Testing Based on Sequence Diagrams” In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Edingburgh, September 2002

[FO00] Fenton, N.E, Ohlsson N. ”Quantitative Analysis of Faults and Failures in a Complex Software System”, IEEE Transactions on Software Engineering, 26(8), pp. 797-814, 2000.

[FP97] Fenton, N.E., Pfleeger S.L. “Software Metrics - A Rigorous and Practical Approach”. Second ed. London: International Thomson Computer Press, 1997.

[FR99] Franch, X., Ribo, J.M. “Using UML for modeling the static part of a software process”.

[FR00] Frestimate, inside the Kevin Tsui and Kelly Chan, SENG609.11 Software Reliability Project: SoftRel, 2000. On line at http://www.softrel.com/prod01.htm

[FY00] Frankl, P.G., Yuetang, D. “Comparison of delivered reliability of branch, data flow and operational testing: A case study”. Proc. of ISSTA 2000. pp. 124-134

[GA92] Gaudoin, O. "Optimal properties of the Laplace trend test for software-reliability models" IEEE Transactions on Reliability, R 41 (4), pp 525-532, 1992

[GA99] Gaudoin, O. "CPIT goodness-of-fit tests for reliability growth models" in Statistical and Probabilistic Models in Reliability, D. C. Ionescu & N. Limnios eds, Birkhaüser, Boston, pp 27-37, 1999.

[GCS95] Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B. “Bayesian data analysis”. Chapman & Hall ,1995.

[GKC01] Garlan, D., Kompanek A.J., Cheng.S.W. “Reconciling The Needs of Architectural Description with Object-Modeling Notations.” Wiley Encyclopedia of Software Engineering, J. Marciniak (Ed.), John Wiley& Sons, 2001.

[G079] Goel., A.L. Okumoto, K. “Time-Dependent Error-Detection Rate Model for Software and Other Performance Measure”. IEEE Transactions on Reliability. Vol. R-28, No.3, August 1979, pp. 206-211

[GOEL] Goel-Okumoto Software Reliability Model Available at http://www.dacs.dtic.mil/about/services/goel.shtml

[GR00] Graubmann, P., Rudolph, E. “HyperMSCs and Sequence Diagrams for use case modeling and testing” "UML" 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference. Proceedings (Lecture Notes in Computer Science Vol.1939), 2000, Pages 32-46

[H323] H.323 Standard. Available at:

http://www.microsoft.com/windows/NetMeeting/Corp/reskit/Chapter11/default.asp

[HIM00] Hartmann, J., Imoberdof, C., Meisenger, M. “UML-Based Integration Testing”. ISSTA 2000, Portland, August 2000.

[HM03] Harel D., Marelly, R. "Specifying and Executing Behavioural Requirements: The Play In/Play-Out Approach", Software and System Modelling (SoSyM), 2003

[HMW01] Hamlet, D., Mason, D., Woit, D. ”Theory of software reliability based on components” Proceedings of the 23rd international conference on Software engineering July 2001. pp. 361-370

[HR01] Herzog, U., Rolia, J. “Performance Validation Tools for Software/hardware Systems”, Performance Evaluation, July 2001.

[HRN01] Host M., Regnell B., Natt och Dag J., Nedstam J., Nyberg C. “Exploring bottlenecks in market-driven requirements management processes with discrete event simulation“. Journal of System and Software Vol 59 pp. 323-332, 2001

[HSM02] NIST/SEMATECH e-Handbook of Statistical Methods, October 2002. Available at: http://www.itl.nist.gov/div898/handbook/, date.

[ICSE03] ICSE ``Workshop on CBSE: Automated Reasoning and Prediction", Portland, Oregon, USA, May 3-4, 2003.

[IKT85] Imai, K., Nonaka, I., Takeuchi, H. “Managing the New Product Development Process: How the Japanese Companies Learn an Unlearn” in Clark, K., b., Hayes, R., H., Lorenz, C. (eds.). The uneasy Alliance. Harvard Business School Press, Boston, 1985.

[IEEE93] IEEE Standard for Software Unit Testing IEEE Std. 1008-1987 (R1993)

[IEEE98] IEEE Recommended Practice for Architectural Description, Draft 3.0 of IEEE P1471, May 1998. Available at http://www.pithecanthropus.com/~awg/

[IEEE98a] IEEE Standard: Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software” IEEE Std. 982.2-1998

[IEEE99] IEEE Computer, July 1999, Vol. 32, No.7.

[IEEE01] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE Std 1471) IEEE Architecture Working GrouP http://www.pithecanthropus.com/~awg

[JBR98] Jacobson, I., Booch, G., Rumbaugh, J. “The Unified Software Development Process”. Addison-Wesley, 1998

[JCJ92] Jacobson, J., Christerson, M. Jonsson, P. Overgaard, G. “Object-Oriented Software Engineering”: A Use Case Drive Approach”. Addison Wesley, 1992.

[JGP98] Jézéquel, J.M, Le Guennec, A., Pennanech, F. ”Validating Distributed Software Modeled with UML”. Proceeding of UML98, in LNCS 1618, pp. 365-376.

[J095] Jorgensen,P. C., “ Software Testing a Craftsman’s Approach”. CRC Press, 1995.

[JSS03] Journal of Systems and Software, Special Issue on CBSE, Volume 65, Issue 3, Pages 169-238 March 2003

[JSW99] Jager D., Schleicher A., and Westfechtel B.: Using UML for Software Process Modeling. ESEC/FSE'99, Toulouse, France, LNCS 1687, Springer, (September 1999).

[JUN] JUNIT tool On-line at: http://www.junit.org

[KA86] Kulkarni, V.G., Adlakha, V.G. “Markov and Markov-Regenerative PERT Networks”. Oper. Res. (1986) Vol. 34, 769-781.

[KCM01] Kim, S.W., Clark, J.A., McDermid J.A. ”Investigating the effectiveness of object-oriented testing strategies with the mutation method” Software Testing, Verification and Reliability, Vol. 11 No.4 pp. 207–225, 2001.

[KCT02] Koppol, P.V., Carver, R.H., Tai. K.C.n “Incremental integration testing of concurrent programs”. IEEE Transactions on Software Engineering, Vol. 28, No 6 pp. 607–623, 2002.

[KFN99] Kaner, C., Falk, J., Nguyen H.Q. “Testing Computer Software”, 2nd Edition, John Wiley & Sons, April, 1999

[KGH95] Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y., Song, Y. ”Developing an Object-Oriented Software Testing and Maintenance Environment”. Communication of the ACM, vol. 32, no. 10, 1995, pp.75-87.

[KHC99] Kim, G.,. Hong, H.S., Bae, D.H., Cha, S.D. “Test Cases Generation from UML State

[KHG02] Kung, D.C, Hsia, P., Gao, J: “Testing Object-Oriented Software” Wiley-IEEE Press October 2002,

[KK00] Kuntzmann-Combelles, A. Kruchten, P. “The Rational Unified Process – An Enabler for Higher Process Maturity “. (Version 1.0 2000). White Paper On line at: http://www.rational.net/rupcenter.

[KKL93] Kanoun, K., Kaâniche, M., Laprie, J.C., and Metge, S., "SoRel: A Tool for Reliability Growth Analysis and Prediction from Statistical Failure Data", FTCS-23 Proceedings, Toulouse, France, June 1993, pp. 654-659.

[KKL97] Kanoun, K., Kaaniche, M., and Laprie, J. P., "Qualitative and Quantitative Reliability Assessment", IEEE Software, Vo. 14, No. 2, March 1997.

[KM90] Korson, T., McGregor, M. "Understanding Object-Oriented: A Unifying Paradigm", Communications of the ACM, pp. 40-60, September 1990.

[KM91] Keiller, P. A., Miller, D. R. "On the use and the performance of software reliability growth models" Reliability Engineering and System Safety, pp. 95-117, 1991

[KMR99] Kellern, M.I., Madachy, R.J., Raffo D.M. "Software Process Simulation Modeling: Why? What? How?" Journal of System and Software. Vol. 46, No. 2/3. April 1999

[KPM] Kerzner Project Management Maturity Online Assessment Tool. Available at http://www.iil.com/brochures/kerzner.htm

[KR95] Kruchten, P. "The 4+1 view model of architecture" IEEE Software. 12(6), November 1995.

[KR98] Kölling, M., Rosenberg, J. "Support for Object-Oriented Testing" in Technology of Object-Oriented Languages and Systems (TOOLS) 28, IEEE, Melbourne, 204-215, 1998

[KR00] Kruchten, P. "The Rational Unified Process: An Introduction". 2nd edition. Addison-Wesley, 2000.

[KU01] Krishnan, V., Ulrich, K. T. "Product Development Decisions: A Review of the Literature" Management Science, (Vol. 47, 2001), 1-21

[LA83] Lavenberg, S.S. "Computer Performance Modeling Handbook". Academic Press, New York, 1983.

[LA92] Laprie, J.C. "Dependability: Basic concepts and Terminology, Dependable Computing and Fault-Tolerant Systems". Vo.l. 5. J.C. Laprie (ed.), Springer-Verlag, Wien, New York 1992

[LA93] Laprie, J.C." Dependability: From Concepts to Limits" Proc. Of SAFECOMP'93, Springer-Verlag, Poznan, Poland, 1993, pp. 157-168.

[LHK02] Lo, J.H., Huang, C.Y., Kuo S.Y., Lyu, M.R.: "Optimal Resource Allocation and Reliability Analysis for Component-Based Software Applications," Proc. 26th Annual International Computer Software and Applications Conference (COMPSAC2002), Oxford, England, August 26-29 2002.

[LI92] Lee. I., Iyer, R.K. "Analysis of Software Halts in Tandem System" Proc. Of ISSRE, October 1992, pp.227-236.

[LI98] Lindemann, C. "Performance Modelling with Deterministic and Stochastic Petri Nets", John Wiley & Sons, 1998.

[LM01] Latella, D., Massink.M. "A formal testing framework for UML Statechart Diagrams behaviors: From theory to automatic verification" In Sixth IEEE International High-Assurance Systems Engineering Symposium. IEEE Computer Society Press, pages 11-22, 2001

[LMR92] Lejter, M., Meyers, S., Reiss, S.P. "Support for Maintaining Object-Oriented Programs," IEEE Trans. Software Eng., Vol. 18, No. 12, Dec. 1992, pp. 1045-1052.

[LN93] Lyu, M.R. , Nikora, A. P.,"CASRE - A computer-Aided Software Reliability Estimation Tool", CASE 92 Proceedings, Montreal, Canada, July 1992, pp. 264-275

[LO87] Loyd, E. “Handbook of Applicable Mathematics: Statistics”. Vol. III, IV, John Wiley & Sons 1987.

[LO98] Loch, CH. “Operations Management and Reengineering”. European Management Journal (Vol.16, 1998), 306 – 317.

[LPM99] Lombardi, G., Peciola, E., Mirandola, R., Bertolino, A., Marchetti, E. “Towards Statistical Control of an Industrial Test Process”. Proceeding of Safecomp '99, Toulouse, September 27-19, 1999, pp. 260-271

[LPS00] Littlewood, B., Popov, P.T., Strigini, L. Shryane N. “Modelling the Effects of Combining Diverse Software Fault Detection Techniques” Transaction on Software Engineering 26(12): 1157-1167 2000

[LPS01a] B Littlewo od, B., Popov, P.T., Strigini, L. "Design Diversity: an Update from Research on Reliability Modelling", Proc. Safety-Critical Systems Symposium 2001, Bristol, UK, Springer-Verlag.

[LPS01b] Littlewood, B., Popov, P.T., Strigini, L. "Modelling software design diversity - a review", ACM Computing Surveys, Vol. 33, No. 2, June 2001, pp. 177-208.

[LRM97] Lyu, M.R., Rangarajan, S., van Moorsel A.P.A. "Optimization of Reliability Allocation and Testing Schedule for Software Systems," Proceedings IEEE ISSRE'97, Albuquerque, New Mexico, November 2-5 1997, pp. 336-346.

[LRM02] Lyu, M.R., Rangarajan, S., van Moorsel A.P.A. “Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development", IEEE Transactions on Reliability, vol. 51, no. 2, June 2002, pp. 183-192.

[LS93] Littewood, B., Strigini, “Validation of Ultra-High Dependability for Software-based Systems”. Communications of the ACM, Vol. 36, No. 11, November 1993, pp. 69-80

[LS00] Littewood, B., Strigini, L. “Software Reliability and Dependability: A Roadmap”. Proc. of the conference on the future of Software engineering 2000, Limerick, Ireland June 2000, pp. 177-188

[LTW00] Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., Durand M. H. “Testing Level for Object-Oriented Software”. Proceeding of ICSE, Limerick, Ireland, June 2000, pp. 136-145

[LV73] Littlewood. B., Verrall. J. “A Bayesian Reliability Growth Model for Computer Software”. Journal of the Royal Statistical Society, series C, Vol22 No. 3, 1973, pp. 331-346.

[LY96] Lyu M. R. (Ed.), “Handbook of Software Reliability Engineering”, McGraw-Hill, 1996.

[LY02] Lyu M. R. ”Software Reliability" Encyclopedia of Software Engineering, Wiley, 2002, pp. 1611-1630.

[LZ99] Liuying. L., Zhichang, Q.: Test Selection from UML Statecharts. Proceeding of 31st International Conference on Technology of Object-Oriented Language and System, Nanjing, China, 22-25 September 1999.

[LZS84] Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik K.C. “Quantitative System Performance Computer System Analysis Using Queueing Network Models” Prentice-Hall, Inc., in 1984. Available at: http://www.cs.washington.edu/homes/lazowska/qsp/

[MA00] Marshall C “Enterprise Modeling with UML: Designing Successful Software through Business Analysis”, Addison-Wesley, (2000)

[MD97] Malaiya, Y. K., Denton, J. A. “What do software reliability parameters represent?” In Proc. International Symposium on Software Reliability Engineering, pages 124–135, Albuquerque, NM, November 1997.

[MEL] M-élopée Évaluation des Logiciels depuis les Phases d'essais jusqu'en Exploitation. Available at http://www.mathix.fr/ http://www.mathix.fr/

[MF96] McFeeley R. "IDEAL: A User's Guide for Software Process Improvement". Software Engineering Institut, Pittsburgh, PA, CMU/SEI-96-HB-001, 1996.

[MFS00] Von Mayrhauser, A., France, R.; Scheetz, M.; Dahlman, E. "Generating test-cases from an object-oriented model with an artificial-intelligence planning system", IEEE Transactions on Reliability, Volume 49, Issue 1, 2000, Pp 26-36

[MGB99] Mercier, F., Le Gall, P., Bertolino, A. "Formalizing integration test strategies for distributed systems". Proceeding of ICSE Workshop Testing Distributed Component-Based System, Los Angeles, May 1999.

[MIO87] Musa, J.D., Iannino, A., Okumoto, K. "Software Reliability – Measurement, Prediction, Application". McGraw-Hill, New York, 1987.

[MJ72] Moranda, P.L., Jelinski, X. "Final Report on Software Reliability Study". McDonnell Douglas Astronautics Company, MADC, Report Number 63921, 1972.

[MO75] Moranda, P.B. "Software Reliability Prediction" Proceeding of the sixth Triennial World Congress of the International Federation of Automatic Control, 1975, pp. 34.1-34.7

[MO83] Musa, J.D, Okumoto, K, "Software reliability models: concepts, classification, comparisons, and practice", Proc. Electronic Systems Effectiveness and Life Cycle Costing Conference, Norwich, U. K., July 19-31, 1982, NATO ASI Series, Vol. F3, (Ed: J. W. Skwirzynski) Springer-Verlag, Heidelberg, 1983, pp. 395-424.

[MPT] Microsoft Project Tool. Available at http://www.microsoft.com/office/project/

[MU93] Musa, J.D. "Operational Profiles in Software-Reliability Engineering", IEEE Software, March 1993, pp.14-32.

[MU96] Musa, J. D., "Software-Reliability Engineered Testing", Proc. of the 9th Int. Software Quality Week, S. Francisco, USA, May 21-24, 1996, paper 2Q2.

[MU98] Musa, J.D. "Software Reliability Engineering: More Reliable Software, Faster Development and Testing, ISBN 0-07-913271-5, McGraw-Hill, New York, 1998.

[MU02] Muccini, H. "Software Architecture for Testing, Coordination and Views Model Checking" PhD Thesis, Università degli Studi di Roma "La Sapienza" 2002. Available at http://www.henrymuccini.com/Research/PhD-Thesis.htm.

[MU03] Musa, J.D. "More Reliable Software Faster and Cheaper (Software Reliability Engineering)" website: http://members.aol.com/JohnDMusa/

[MW82] Martw, H.F., Waller, R.A. "Bayesian Reliability Analysis", New York, John Wiley & Son 1982

[MZ98] Mitchell, B., Zeil, S.J. "An Experiment in Estimating Reliability Growth Under Both Representative and Directed Testing" Proc of ISSTA 1998.pp 32-41

[NAS97] NASA, "Formal Methods, Specification and Analysis Guidebook for the Verification of Software and Computer Systems". Vol. II. A Practitioner's Companion, NASA GB-Oo1.97, 1991. http//eis.jpl.nasa.gov/quality/Formal_Methods/.

[NET] .Net resources available at: http://www.microsoft.com/net/

[NFF03] Neil M, Fenton N, Forey S, Harris R. "Assessing Vehicle Reliability using Bayesian Networks" in Global Vehicle Reliability, Edited by J. E. Strutt and P.L. Hall. Professional Engineering Publishing, 25-42, 2003.

[NKF03] Neil M, Krause P, Fenton NE, "Software Quality Prediction Using Bayesian Networks" in Software Engineering with Computational Intelligence, (Ed Khoshgoftaar TM), Kluwer, ISBN 1-4020-7427-1, Chapter 6, 2003

[NLS02] Di Nitto, E., Lavazza, M., Schiavoni, Tracanella, E., Trombetta M. "Deriving executable process descriptions from UML". ICSE 2002, Orlando, Florida, (May 2002).

[OA99] Offutt, J., Abdurazik, A. "Generating Test from UML Specifications". Proceeding of UML 99, Fort Collins, CO, October 1999.

[OA00] Offutt, J., Abdurazik, A.: Using UML Collaboration Diagrams for Static Checking and Test Generation. UML 2000, University of York, UK, 2-6 October 2000.

[OB88] Ostrand, T.J., Balcer, M.J. "The Category Partition Method For Specifying and Generating Functional Tests". Communication of the ACM, vol. 31, no.6, June 1988, pp. 676-686.

[OBJ] Objecteering Software. On line at http://www.objecteering.com/

[OH84] Ohba, M. "Software Reliability Analysis Models". IBM Journal of Research and Development, Vol. 21. No. 4, July 1984, pp 428-443

[PCC93] Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V., "Capability Maturity Model, Version 1.1", IEEE Software, Vol. 10, No. 4, July 1993, pp.18-27

[PE95] Perry, W. "Effective Methods for Software Testing", Wiley 1995

[PF98] Pfleeger, S., L., "Software Engineering Theory and Practice", Prentice Hall, 1998

[PJC97] Pfleeger, S.L., Jeffery, R., Curtis, B., Kitchenham, B. "Status Report on Software Measurement". IEEE Software Vol.14 No.2 1997 pp. 33-44.

[PJT02] Pickin, S., Jard, C., Le Traon, Y. Jéron, T., Jézéquel JM., Le Guennec, A. " System test synthesis from UML models of distributed software" In D. Peled and M. Vardi, editors, Formal Techniques for Networked and Distributed Systems - FORTE 2002, LNCS, Houston, Texas, November 2002

[PMB99] Powell A.L., Mander K.C.,Brown D.S. "Strategies for Lifecycle Concurrency and Iteration - A System Dynamics Approach," Journal of Systems & Software, Vol. 46 N.2--3, 1999.

[PO02] Popov, P., "Reliability Assessment of Legacy Safety-Critical Systems Upgraded with Off-the-Shelf Components", SAFECOMP'2002, September 2002, Catania, Italy

[PS02] Petriu D.C., Shen H. "Applying the UML Performance Profile: Graph Grammar-based derivation of LQN models from UML specification". Proceedings of Performance TOOLS 2002, London, England, April 14-17 2002, LNCS 2324, Springer Verlag

[POS] Poseidon tool. Available at http://www.gentleware.com

[PR94] Pressman, R., S., "Software engineering: a practitioner's approach", McGraw-Hill Book Company Europe, 1994

[PRIS] PRISM is in Reliability Analysis Center (RAC). On line at http://rac.alionscience.com/prism/

[PSM02] Popov, P., Strigini, L., May, J., Kuball, S. "Estimating Bounds on the Reliability of Diverse Systems", IEEE Transactions on Software Engineering, 2002.

[PW92] Perry D.E., Wolf A.L. "Foundations for the Study of Software Architecture". ACM SIGSOFT Software Engineering Notes, 17(4): 40-52, October 1992.

[PW93] Peng, W., W., Wallace D., R., "Software Error Analysis", NIST SP 500-209, National Institute of standards and Technology, Gaithersburg, MD 20899, December 1993, http://hissa.nist.gov/SWERROR/

[RA00] Ramaswamy R. "How to Staff Business Critical Maintenance Projects". IEEE Software Vol. 7, 2000, pp. 90-95.

[REMA] Reliability & Maintenance Analyst Available at http://www.engineeredsoftware.com/rma.asp

[RG00] Ryser, J., Glinz M. "Using Dependency Charts to Improve Scenario-Based Testing". Proceedings of the 17th International Conference on Testing Computer Software (TCS2000). Washington D.C., June 2000.

[RGK00] Ramani, S. Gokhale, S.S., Trivedi K.S. "SREPT: Software Reliability Estimation and Prediction Tool Performance Evaluation, Vol. 39, pp. 37-60, 2000.

[RJB99] Rumbaugh J., Jacobson I., Booch J. "The Unified Modeling Language Reference Manual" Addison Wesley, 1999.

[RPG02] Riebisch, M. Philippow, I.;Götze, M. "UML-based Statistical Test Case Generation" Net.Object.Days 2002 Erfurt, Germany, October 7-10, in Lecture Notes in Computer Science Vol. 2591, pp 394-411, 2002

[RRT] Rational Rose tool. On line at http://www.rational.com/products/rose/index.jsp

[RS01] Rosemberg, D., Scott, K. "Applying Use Case Driven Object Modeling With UML: An annotated E-Commerce Example". Addison Wesley June 2001

[RSC] Rational Software Corporation's. http://www.rational.com/

[RUP] Rational Unified Process version 2000.02.10. Rational Software Corporation. On-line at http://www.rational.com/products/rup

[SBA01] Saleh, D., Boujarwah, A.A., Al-Dallal J. "Anomaly detection in concurrent Java programs using dynamic data flow analysis". Information and Software Technology, Vo.43 No 15 pp 973–981, 2001.

[SC75] Schneidewind, N.F. "Analysis of Error Processes in Computer Software" Sigplan Note, Vol. 10, no. 6, 1975, pp. 337-346

[SC99] Schneider S.A. "Concurrent and Real-time Systems: the CSP approach". John Wiley, New York, 1999.

[SCK01] Seo, H.S., Chung, I.S., Kim, B.M, Kwon. Y.R. "The design and implementation of automata-based testing environment for Java multi-thread programs". In proceeding of the 8th Asia Pacific Software Engineering Conference (APSEC '01), Los Alamitos, California, 2001 pp. 221–228.

[SD01] Smith G., DerrickJ.M. "Specification, refinement and verification of concurrent systems: an integration of Object-Z and CSP". Formal Methods in System Design, Vol. 18, No. 3 pp. 249–284, 2001.

[SE01] Selic, B. "Response to the OMG RFP for Schedulability, Performance and Time" OMG document Ad/2001-06-14.

[SE02] Selic, B. "Performance-Oriented UML capabilities" Proceedings of Third International Workshop on Software and Performance, Roma, Italy, July 2002, ACM press.

[SH72] Shooman, M.L. "Probabilistic Models for Software Reliability Prediction". Statistical Computer Performance Evaluation, Academic Press, New York, June 1972, pp 485-502

[SM90] Smith, C.U. "Performance Engineering of Software Systems". Addison-Wesley, Reading, MA, 1990.

[SM97] Smith, R. "The Historical Roots of Concurrent Engineering Fundamentals". IEEE Transaction on Engineering Management (Vol. 43, 1997), 67-78.

[SME96] SMERFS; Statistical Modeling and Estimation of Reliability Functions for Systems. Available at http://www.slingcode.com/smerfs/. Copyright 1996

[SOR93] SoRel: Software Reliability Available at: http://www.laas.fr/surf/sorel/sorel.html. Copyright 1993

[SP99] Spivey, J. M. "The Z notation: A reference manual2. Prentice-Hall, 1989.

[SPM] Software Program Manager's Network. On line at http://www.spmn.com.

[ST97] Stieber, H.A. "Statistical Quality Control: How to Detect Unreliable Software Components". In Proceedings of the 8th International Symposium on Software Reliability Engineering, Albuquerque, NM, USA, November 1997, published by IEEE Computer Society

[ST00] Stieber, H.A. "Optimal Testing Strategies" In Proceedings of the 10th International Symposium on Software Reliability Engineering, San Jose, CA, USA, October 2000.

[SW01] Smith, C.U. and L. Williams. Performance Solutions: a Practical Guide to Creating Responsive, Scalable Software", Addison-Wesley, 2001.

[TN86] Takeuchi, H., Nonaka, I. "The New Product Development Game". Harvard Business Review (Vol. 64, 1986), 137-146.

[TSP99] Tsai, B.Y.; Stobart, S., Parrington, N., Mitchell, I. "Automated class testing using threaded multi-way trees to represent the behavior of state machines", Annals of Software Engineering, Volume 8, 1999, Pages 203-221

[UML] UML Documentation version 1.5 Web Site. On-line at http://www.omg.org/technology/documents/formal/uml.htm

[UML00] Evans, A., Kent, S., Selic B., (Eds.): «UML» 2000 - The Unified Modeling Language. Advancing the Standard Third International Conference, York, UK, October 2000. Proceedings LNCS 1939, Springer Verlag

[UML01] Gogolla, M., Kobryn C., (Eds.): UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings LNCS 2185, Springer Verlag.

[UML02] Jézéquel, J.M., Hussmann, H., Cook c., (Eds.): UML 2002 - The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings LNCS 2460, Springer Verlag

[UMLA] UMLAUT Available at http://www.irisa.fr/UMLAUT/

[UMLP] UML™ Profile for Schedulability, Performance, and Time Specification. On line at htttp://www.omg.org/cgi-bin/doc?ptc/02-03-02.pdf

[VJB03] Vegas S. Juristo N. Basili V.R., "Identifying Relevant Information for Testing Technique Selection: An Instantiated Characterization Schema" The Kluwer International Series in Software Engineering, Kluwer Academic Publishers, Boston Volume 8 April 2003.

[WC99] Williford J., Chang A. "Modeling Fed Ex's IT Division: A System Dynamics Approach to Strategic IT Planning". Journal of Systems & Software, Vol. 46 N.2-3, 1999.

[WE86] Weiss, G. "Stochastic Bounds on Distribution of Optimal Value Function with Application to PERT, Network Flows and Reliability", Oper. Res. (Vol. 36, 1986), 595-605.

[WH92] Wilde, N., Huitt, R. "Maintenance Support for Object-Oriented Programs" IEEE Trans. Software Eng., Vol. 18, No. 12, Dec. 1992, pp. 1038-1044.

[WI99] Williams, C.E. “Software Testing and the UML", Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton, November 1-4, 1999.

[WK99] Warmer, J., Kleppe, A. “the Object Constraint Language” Addison-Wesley, 1999]

[WM01] Wittevrongel, J. Maurer, F. “Using UML to Partially Automate Generation of Scenario-Based Test Drivers”. OOIS 2001, Springer, 2001

[WO97] Wood. A. “Software Reliability Growth Models: Assumptions vs. Reality” Eighth International Symposium on Software Reliability Engineering (ISSRE '97), November 02 - 05, 1997,Albuquerque, NM. pp. 136

[WO00] Wosp2000, Proceedings of Second International Workshop on Software and Performance, Ottawa, Canada, September 2000, ACM press.

[WO02] Wosp2002, Proceedings of Third International Workshop on Software and Performance, Roma, Italy, July 2002, ACM press.

[WR01] Wallace, D. Reeker, L. “Knowledge Area Description of Software Quality”. Chapter 11 of SWEBOK.: The Guide to the Software Engineering Body of Knowledge. Joint IEEE-ACM Software Engineering Coordination Committee. (2001). On line at http://www.swebok.org/.

[XHW97] Xie., M., Hong, G.Y., Wohiln, C. “A Practical Method For The Estimation Of Software Reliability Growth In The Early Stage Of Testing” Eighth International Symposium on Software Reliability Engineering (ISSRE '97) Albuquerque, NM, November 02 - 05, 1997 p. 116

[YLK02] Yu, C.Y., Lo, J.H., Kuo, S.Y., Lyu, M.R. “Optimal Allocation of Testing Resources for Modular Software Systems” Proc. 13th International Symposium on Software Reliability Engineering (ISSRE 2002), Annapolis , MD , November 12-15 2002

[YOO83] Yamada, S., Ohba, M., Osaki S. “S-Shaped Reliability Growth Modeling: Models and Assumption”. IEEE Transaction on Software Engineering. Vo. SE-11, No. 12, December 1985, pp 1431-14237

[ZHM97] Zhu, H., Hall, P.A.V., May, J.H.R “Software Unit Test Coverage and Adequacy”. ACM Computing Surveys, 29, 4 Dec. 1997, pp. 366-427