ICEBERG

How to estimate costs of poor quality in a Software QA project: a novel approach to support management decisions

Industry-Academia Partnerships and Pathways (IAPP)
Call: FP7-PEOPLE-2012-IAPP

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n°324356

Deliverable No.:	***3.1***
Deliverable Title:	*First measurement/prediction models-based process*
Organisation name of lead Contractor for this Deliverable:	*CINI, UAH*
Author(s):	*R. Pietrantuono, P.Potena, L.Fernandez,D. Rodriguez*
Participant(s)	*All*
Work package contributing to the deliverable:	*3*
Task contributing to the deliverable:	*T 3.1*
Total Number of Pages	*XX*

Table of Versions

Version	Date	Version Description	Contributors
0.1	28-02-2014	Draft. First document skeleton.	Roberto Pietrantuono
0.2	13-03-2014	Added Workflow.	Pasqualina Potena, Luis Fernandez
0.3	28-03-2014	Added models description: release planning model, resource allocation model, defect analysis model	Roberto Pietrantuono
0.4	07-05-2014	Refined description of models for: release planning, resource allocation, defect analysis	Roberto Pietrantuono
0.5	31-05-2014	Final description of models and process, refining selection of models, definition of parameters and feasibility analysis according to existing information on real industry practices	Pasqualina Potena, Roberto Pietrantuono, Daniel Rodríguez, Luis Fernández, Makalay Yatsevich

Contents

1 EXECUTIVE SUMMARY

The aim of D3.1 of the ICEBERG project "First measurement/prediction models-based process" is to describe the methodological models-based process under development. The document describes the first version of this process, which will be enriched/refined in the course of the project once the subsequent phases provide data and additional checks on the feasibility, precision and effectiveness of it for practice in real scenarios in project decision making. The goal of our work is to define measurement/prediction models able to determine the cost of quality (and not-quality) and allow finding the best trade-off between cost and quality, and the process formalization based on such models. The document describes the workflow followed for defining the process, and provides detailed description of the adopted models so far.

2 INTRODUCTION

In order to support project management decisions on quality assurance actions, the ICEBERG project foresees an approach based on models. The objective is to define a set of models capable of exploiting historical data in order to create a systematic knowledge base and perform quantitative evaluations of the effect of decisions from the quality, cost, and time/schedule point of view. As discussed in previous ICEBERG documents [1] [2], there are several categories of decisions that might impact the quality/cost/time factors of a project. For each (class of) decision(s), one or more models can be used to support project managers.

The choice of which decisions to support depends on what type of historical data a company is expected to gather in its process, what is the cost of collecting such data, and what is the expected benefit (thus the impact) on the process in terms of quality/time/cost trade-off. There may exist a perfect model allowing one to obtain the best trade-off as output, but that requires a great bunch of data as input parameters. Therefore, in the ICEBERG project we adopt an approach that proposes a set of models starting from the actual need of the industry, basing the choice on the availability of the information they can rely on.

This document describes the model-based process under development in the ICEBERG project. It is the first version of this process, which will be enriched/refined in the course of the project once the subsequent phases provide data and additional checks on the feasibility, precision and effectiveness of it for practice in real scenarios in project decision making. The document first describes the workflow followed for defining the process (Section 3); then, an overview of the process is provided (Section 4), followed by a description of the use of quality frameworks in practice (Section 5), and a detailed description of the adopted models so far (Section 6, 7, and 8). Section 9 concludes the document.

3 WORKFLOW FOR THE MODELS-BASED PROCESS DEFINITION

In order to get to a sound process definition, the ICEBERG project followed the workflow depicted in Figure 1. As first step, we analysed the existing works that have been proposed in the literature and the current practices in the industry settings regarding the approaches to support the quality decision making process. An analysis of the State Of The Art (SOTA) and of the State Of The Practice (SOPA) concerning quality decision-making, costs and schedule/time factors (*Step 1* and *Step2*) was carried out. This is the basis for defining the prototypal framework for quality decision-making based on cost, quality and schedule/time trade-off (*Step 3*). This latter step will be performed by exploiting: (i) the identified costs, schedule/time and quality parameters, and (ii) the existing techniques and methods for model building and model solving.

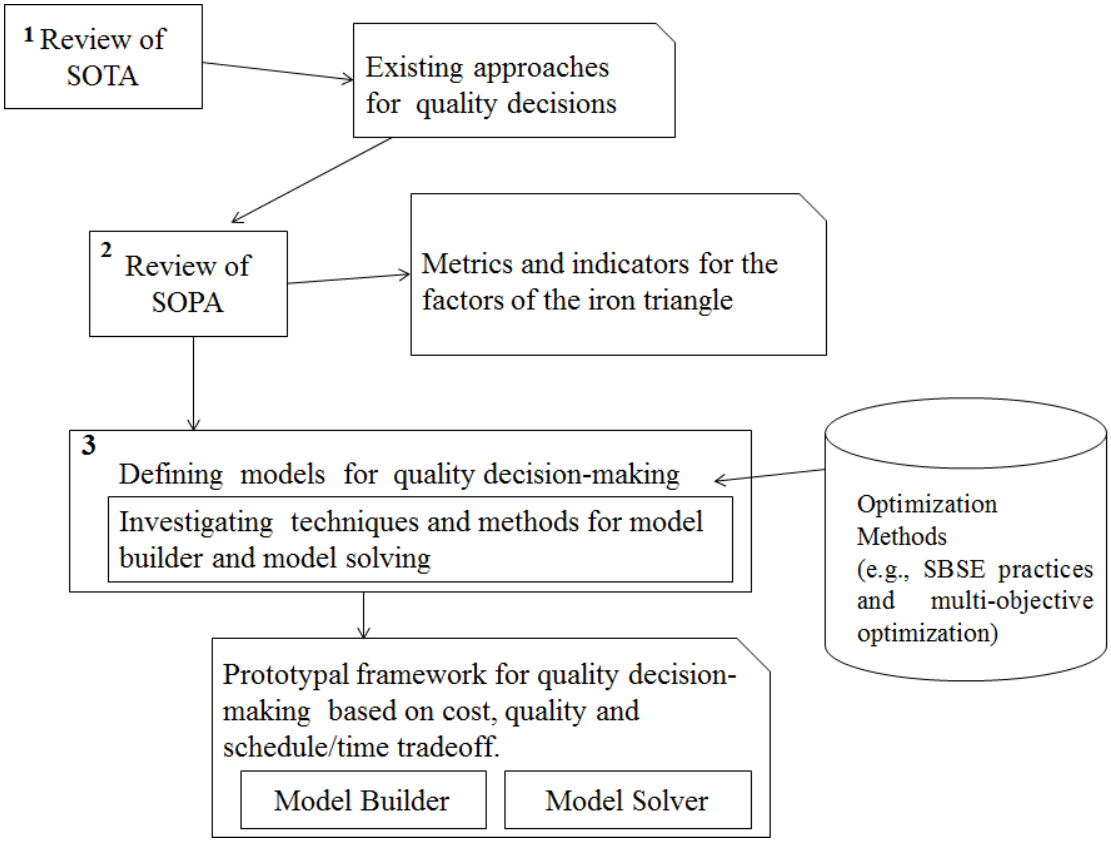

Figure 1: Work Steps

Specifically, these are high-level work's goals (i.e., long term objectives) that we intend to achieve. We will refine these high level goals into more concrete sub goals (i.e., short term objectives) until it is possible to objectively measure their satisfaction. Below, a more detailed description of the main steps follows.

Step 1: Review of SOTA. The ICEBERG project aims at defining how project management decisions on quality assurance actions influence project's results in terms of intrinsic product quality while evaluating their effects in costs and schedule, following the idea of the Iron Triangle for project management.

The ability to predict (or estimate) the software quality supports a large set of decisions across multiple lifecycle phases that span from design through implementation-integration to adaptation phase. However, due to the different amount and type of information available, different prediction/estimation approaches can be introduced in each phase. A major issue in this direction is that the software quality cannot be analyzed separately, because the project managers must assure the respect to constraints on schedule and costs. A quality decision, for example, can be the one of implementing static code analysis (e.g. tools, new processes, training, etc.) but its impact on project schedule, for example, can cause delays in completion of projects tasks while number of defects might be reduced up to certain extent leading to cost savings: in the end, the project manager needs to know if this is helpful and convenient for the project goals.

Therefore, we analysed the existing solutions that concern quality decision-making, cost and schedule/time issues all along the software lifecycle in order to understand:

- What approaches have been reported regarding quality decision-making in the single lifecycle phases?
- Which the quality decisions are (e.g., adaptation mechanisms typically used in adaptive systems) treated in the scientific literature? What is the relationship between these identified decisions and software defects (incidents and other concepts)?
- What are the common causes of decision-making (such not satisfying of constraints on reliability)?
- What approaches have been reported dealing with human and organizational factors? For example, how do the approaches deal with the problem of automating and optimizing shift allocations to people in order to meet certain service levels?
- Which are the schedule/time/cost-related properties considered by the existing approaches for quality decision-making? What is the importance of these identified properties? What is the relationship between these identified properties and the software quality?

Results of this analysis are reported in the previous deliverables (deliverable D2.1 and D2.2).

Step 2: Review of SOPA In order to determine measures for assessing the three factors (i.e., cost, schedule/time and quality), we planned and distributed an interview-survey in which several customer of project's partners have been involved. Our study is based on a method specified in [3]. As explained in the deliverable D2.2, the survey will be conducted in multiple stages that span from questionnaire preparation through data collection and analysis to validity addressing. This information is essential to validate which options from SOTA could be feasible for practitioners according to SOPA. Decisions and the corresponding models selected for the first version of the model-based process, reported hereafter, are based on data and/or information collected from the ICEBERG project partners; the results of the survey will be used to enrich the set

of models and refine the existing ones by considering the metrics/data as collected by the involved company, and the way they collect such information.

Step 3: Defining frameworks for quality decision-making. The aim of this step is to define the prototypal framework for quality decision-making based on cost, quality and schedule/time tradeoff. The framework will be based on the usage of a set of models, orchestrated in a flow taking the product/process information as input and providing solutions to crucial decisions along the development process (e.g., for architectural design, for testing, for debugging, etc.) in terms of quality/cost/schedule prediction/estimation.

4 MODELS-BASED PROCESS OVERVIEW

Figure 2 shows an example of a prototypal framework within its working environment. In this case, the framework comprises two modules: a *Model Builder* and a *Model solver*.

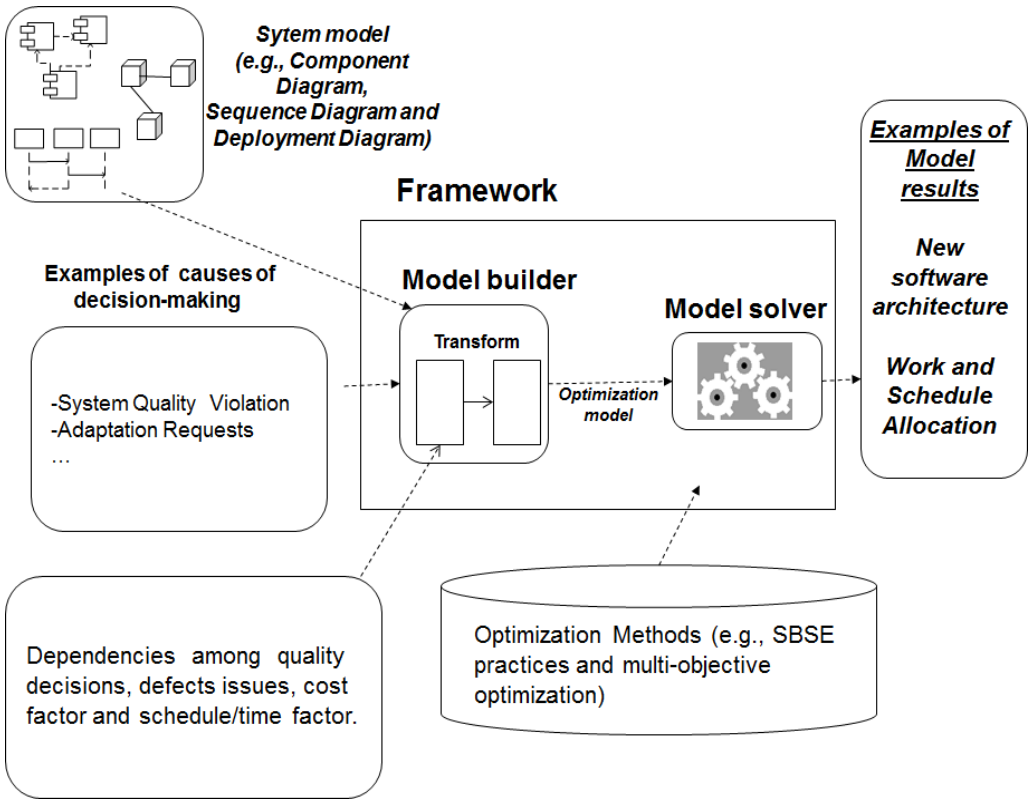

Figure 2: An example of framework and its environment

A primary input to this framework is represented, for example, by (i) the system models (e.g., an UML-based architectural model composed of a Component Diagram, Sequence Diagrams, and a Deployment Diagram), (ii) the causes of quality decision-making, and (iii) dependencies among quality decisions, defects issues, cost factor and schedule factor. In particular, we intend to categorize the identified: (i) quality decisions (and causes), and (ii) schedule/time/cost-related properties. This should be based on results of tasks WP 2.4 and WP 2.5.

The Model Builder generates the optimization model in the format accepted from the solver. The Model solver processes the optimization model received from the builder and produces the results, which consist of a set of quality decisions. It suggests, for example, how to design (or re- design) the software architecture in order to minimize the costs while keeping the software quality within a given threshold. In addition, the model, for example, could also suggest the best shift allocations to people in order to achieve the required level of software quality. The inferences and relationships detected for this model should be created by defining and applying the most appropriate methods for data analysis. Any combination of quality decisions may have a considerable impact

on the cost, time and software quality. Therefore, the optimization model aims to quantify such impact in order to suggest the best quality decision, which minimizes the costs while satisfying the schedule/time, cost and quality constraints.

In order to achieve the right tradeoff among schedule/time constraints, software qualities and costs requirements, the quality decisions should involve the evaluation of new alternatives to the current (i) software application level (e.g., by the configuration of software components, the introducing new components into the system, etc.) and (ii) project management level (e.g., the shift allocations to people). A decision, for example, taken for modifying a system functionality may be good for the satisfaction of a certain level of software quality, but at the same time it may require a high cost for implementing static code analysis (e.g. tools, new processes, training, etc.). A major challenge is then finding the best balance among many different competing and conflicting constraints.

For these multi-attribute problems, there is usually no single global solution, and the generation and evaluation of quality decisions alternatives can be error-prone and lead to suboptimal decisions, especially if carried out manually by system architects or maintainers.

In order to address such problems, we will investigate the application of: (1) SBSE search methodologies (e.g., genetic algorithms, evolutionary algorithms and other metaheuristics) and, (2) the multi-objective optimization, where objectives represent different properties (e.g., cost, time and other software quality-related). Specifically, we will devise a set of solutions, called Pareto optimal solutions or Pareto front, each of which assures a tradeoff between the conflicting constraints. In other words, while moving from one Pareto solution to another, there is a certain amount of sacrifice in one objective(s) to achieve a certain amount of gain in the other(s). Each point of a Pareto curve will be a chain of quality decisions (leading changes either to the application level or the project management level).

In the past five years SBSE has proved to be a widely applicable and success-full approach. In fact, it has been applied to several problems throughout the software engineering lifecycle, from requirement and project planning/management to maintenance and reengineering. In particular, SBSE potential has been already proposed and used for supporting both the software application level and project management level. The SBSE approach results attractive because it provide a suite of adaptive automated and semi automated solutions in situations typified by large complex problem spaces with multiple competing and conflicting objectives [4].

5 USE OF FRAMEWORKS FOR QUALITY DECISIONS IN PRACTICE

A Decision Support System (DSS) is a computer-based information system that supports business or organizational decision-making activities. DSSs serve the management, operations, and planning levels of an organization (usually mid and higher management) and help to make decisions, which may be rapidly changing and not easily specified in advance (Unstructured and Semi-Structured decision problems). Decision support systems can be either fully computerized, human or a combination of both. [1]

A taxonomy for DSS has been created by Dah and Stain in [5]. They differentiate in *model-driven DSS* and *data-driven DSS*. Power in [6] extended this classification by considering *document-driven DSS, communication-driven DSS, and knowledge-driven DSS.*

*Model-driven DSS*s usually provide a mathematical model, based on statistical (optimization or simulation). This model helps make decisions. A user interface is typically used. Such interface facilitates the use of the model.

Data-driven DSSs, also called data-oriented, emphasize access and manipulation of internal and external organizational data (usually numerical). It is not provided a mathematical model. However, the consultation of the data or their temporal is supported. These types of DSSs involve aspects of databases managements systems (e.g., *data warehouses* or *data warehouse*, the online analytical processing (OLAP) and data mining or ETL, etc.).

Data mining techniques can be grouped into predictive and descriptive depending on the problem at hand. From the predictive point of view, patterns are found to predict future behaviour. In fault prediction, it would correspond to the generation of classification models to predict whether a software module will be defective based on metrics from historical project data. From the descriptive point of view, the idea is to find patterns capable of characterising the data represented in such a way that domain experts can understand them (e.g., rules or decision trees).

Considering the objective, data mining task are typically categorised as:

- Classification, prediction task which tries to assign a new instance to a predefined category, e.g., defect classification.
- Regression, typically considered when the output model is a number, e.g., effort or cost estimates.
- Clustering, the objective is to group similar data (e.g., similar open source packages to find alternatives, etc.).

[1] Decision Support Systems - http://en.wikipedia.org/wiki/Decision_support_system

- Time series analysis, evolution of some variables with respect to the time variable (e.g, complexity of a module, accumulated number of modifications, etc.).

- Text mining, the objective is to extract knowledge from free form text, for example, requirements or bug reports could be automatically classified after some text mining preprocessed.

These are object can be achieved using different representation models (techniques) such as trees, rules, artificial neural networks, etc. These models can be categorised as blackbox or whitebox models. Blackbox models include neural networks or support vector machines. Whitebox techniques include rules (for both association rules or classification) or decision trees which are simple to use and provide an explication about the *decision*.

Data mining in Software Engineering

Currently, software organisations produce a large amounts of data from configuration management systems (software), bug tracking systems, mailing lists, etc. These data need to be preprocessed and analysed

Data mining in software engineering has its own challenges [7], [8], [9], [10] as techniques from Web Mining, Text Mining, etc. need to applied and adapted. A subarea of this field is known as SBSE[2], which deals with the application of search and metaheuristic techniques in SE and has become an important area of research (as explained in Section 4). Many SBSE problems are composed of one or more fitness functions that evaluate a search space, which can be generated while searching for the solution or from repositories forming a combinatorial problem from dataset attributes. Its importance is also reflected in several conferences and workshops such as Mining Software Repositories (MSR)[3], PRedictOr Models In Software Engineering (PROMISE)[4], Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)[5].

Data Source in Software Engeneering

Data to apply data mining can come from the following sources

- Documentation of a project, both internal and external. The format can vary a lot and these will need extensive preprocessing and probably text mining in many cases.

- Design, testing, quality assurance information

[2] http://crestweb.cs.ucl.ac.uk/resources/sbse repository/

[3] http://msrconf.org/

[4] http://promisedata.googlecode.com/

[5] http://promisedata.org/raise/

- Source code
- Compiled source, execution traces, logs, etc.
- Bug tracking systems

Software Engineering Repository Software projects leave a trail in different kinds of repositories, and this trail can be used to reconstruct the history of the project, and to study the software development and maintenance processes. We classify this trail in the following categories:

- Source code

This is the most obvious product of a software project. Source code can be studied to measure its properties, such as size or complexity.

- Source Code Management Systems (SCM)

SCM repositories make it possible to store all the changes that the different source code files undergo during the project. Also, SCM systems allow for work to be done in parallel by different developers over the same source code tree. Every change recorded in the system is accompanied with meta-information (author, date, reason for the change, etc) that can be used for research purposes.

- Issue tracking systems

Bugs, defects and user requests are managed in issue tracking systems, where users and developers can fill tickets with a description of a defect found, or a desired new functionality. All the changes to the ticket are recorded in the system, and most of the systems also record the comments and communications among all the users and developers implied in the task.

- Messages between developers and users

In the case of free / open source software, the projects are open to the world, and the messages are archived in the form of mailing lists, which can also be mined for research purposes. There are also some other open message systems, such as IRC or forums. Other projects which are developed in public can also store messages, but it is unusual to have that information for research purposes.

- Meta-data about the projects

As well as the low level information of the software processes, we can also find meta-data about the software projects which can be useful for research. This meta-data may include intended-audience, programming language, domain of application, license (in the case of open source), etc.

- Usage data

In the case of the user side, the trail that projects leave is virtually invisible. There are statistics about software downloads on the Internet, but that is not the only way users get their software. Some of the research datasets we describe in this paper include information about usage data, which is recorded thanks to the collaboration of users.

Testing and Debugging is most prolific research field in which machine learning is being applied. Tools related to testing tools are also the ones have included optimisation or machine learning capabilities, for example, PEX5[6] with the automation of unit tests.

We next briefly describe works according to their research field

Project Management One area of project management is related to the estimation of effort, cost which is mainly a regression problem. Recently data mining techniques have been applied such as Neural Networks and Genetic Programming [11], Case-based Reasoning [12], etc. Another problem related to project management is the allocation of resources and order of activities within a project or across multiple projects. In this case, optimisation and meta-heuristic techniques have also been implemented (e.g. [13] [14]).

Software Requirements This field of research it has been mainly tacked as an optimisation problem, e.g., ranking requirements. Other [15]

Design and Implementation A survey [16]

Testing and Software quality This field is the one that has attracted more attention. Most of the literature regarding data mining and software engineering is related to software quality in the sense of defect prediction (comprehensive surveys include the work of Hall et al. [17] and Catal and Diri [18]), test case generation [19], etc.

Maintenance There are some works related to maintenance effort (e.g. [20]) or automatic repairing of code (e.g. [21]), refactoring (e.g. [22]).

An example of use of a quality decision framework

The goal of quality decision framework is to provide support for the decisions that project managers (developers or testers) take.

Figure 3 schematically delineates the use of the quality decision-making framework. The framework, for example, will suggest the best actions to taken according to a set of new requirements to be satisfied. The framework will take into account cost, schedule and quality factors. It will aim, for example, to minimize the costs while keeping the reliability and the performance of the software architecture within certain thresholds.

[6] http://research.microsoft.com/en-us/projects/pex/

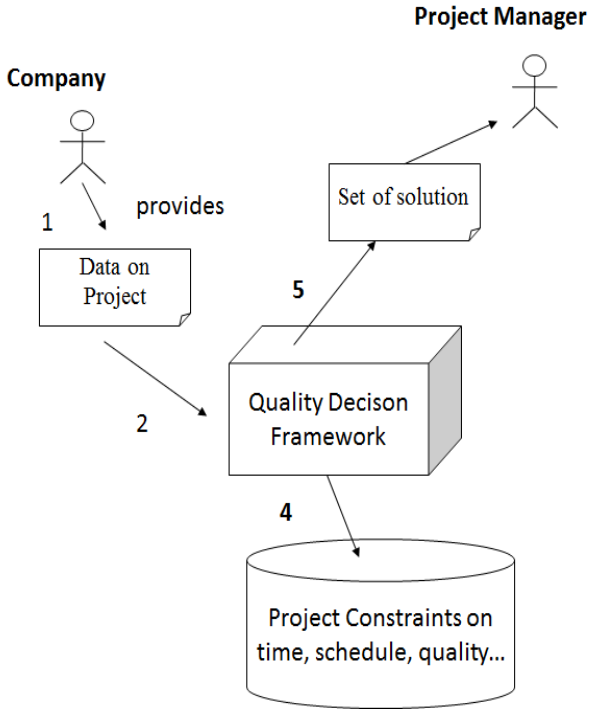

Figure 3: An example of framework application

A primary input to the framework is represented by data collected from company composed of, for example: (i) bugs; and (ii) software metrics. The framework will suggest the best "actions" (e.g., the additional amount of testing to be performed) under quality, cost, and schedule tradeoff.

6 QUALITY DECISION MAKING APPROACHES

As remarked in the deliverable D2.2, several research efforts have been devoted to the definition of quality decision-making approaches in each phase of the software lifecycle. The analysis of system qualities supports a large set of decisions across multiple lifecycle phases that span from design through implementation-integration to adaptation phase. However, due to the different amount and type of information available, different quality decision frameworks can be exploited in each phase.

The goal of our work is to provide support for the decisions that practitioners take. We aim at enabling practitioners to maximize the effectiveness of their specific software by exploiting guidelines of existing literature studies. As explained in the deliverable D2.2, we are collecting information on which types of decisions are normally made by managers or project leaders during the projects. We aim to understand practitioners perceived strengths, limitations, and needs associated with using SOTA practices in the industry.

On one hand, we believe that well-assessed optimization methods (like SBSE methodologies combined with multi-objective optimization), and software quality validation techniques, will helpful to assist software designers/maintainers and software project managers during the whole software system lifecycle.

On the other hand, we plan to analyze effort and time necessary to incorporate the SOTA solutions into real-world systems: as intended in the plan of Iceberg project, this is going to be addressed with some industrial scenarios provided by industrial partners, namely Assioma.net and DEISER.

As explained in the deliverable D2.2, we can see from the first questionnaire's results that practitioners usually deal with a few software metrics, or defect (cost, schedule, and time) data. We will investigate the effort required for collecting additional data (e.g., additional software metrics or particular cost factors, such as that of test cases generation). Because we have pointed out that practitioners already use some tools for source code metrics evaluation (e.g., Sonar), and bug tracking (e.g., JIRA), or they are willing to adopt them in their company, we believe that these kinds of software metrics could be easy collected and evaluated. We will analyze which is the effort required for using in industry the tools for software metrics evaluation.

We have also realized that the testing is a typical activity in the industry, and practitioners are willing to invest to improve it. We think that testing is also a good "provider" of data for the cost, schedule and time indicators. Therefore, we will investigate, for example, (i) which are the main features of these tools adopted for the testing, and (ii) how these tools could be integrated with other tools (e.g., the ones for cost or time evaluations) in order to obtain both cost and time data (such as, for example, that of test cases generation or execution).

In this section, we describe existing SOTA studies, which might be helpful to enable practitioners to maximize the effectiveness of their specific software. These quality-decision making approaches can be categorized by the kind of techniques used: (a) *defect analysis model-based*, and (b) *optimization model-driven*.

Defect analysis model-based approaches basically give guidelines to evaluate the development process by considering software metrics and defect data. The purpose of defect analysis is to have quantitative support for evaluating the development process. Depending on the grain of the analysis, the supported quality decisions may regard: the improvement of the most critical phases of the development process, improvement actions on the most critical components, actions against the most critical suppliers, and/or the most critical actors (e.g., testing teams) involved in the process. We are developing a lightweight defect analysis approach, merging various types of analyses and models to both product and process evaluation, which requires a minimal set of defect information as input in order to keep the impact on current practices low.

The *optimization model-driven* category mainly encompasses studies that, besides providing a means to evaluate system quality, support the making of best quality decisions.

This section reports the models that are currently under development for supporting decision making process, with a specific impact on the quality of the product. We discuss the adopted models for defect prediction, focusing on identifying the most critical modules and thus supporting testing. Additionally, we devise the formulation of cost minimization problems under quality constraints. These optimization models stem from our previous works, where we have instantiated them for the phases of design and adaptation. However their elements (e.g., cost function and reliability/performance constraints) could be re-used in another phase of the software life cycle phase. We are investigating how to reuse the guidelines of these existing models in order to allowing study of the tradeoffs among quality, cost, and time attributes.

In Section 6.1, we describe defect analysis model-based approaches, and, in Section 6.2 and Section 6.3, we provide the guidelines of our optimization models.

6.1 DEFECT ANALYSIS MODEL-BASED APPROACH

Supported QA decisions

The purpose of defect analysis is to have quantitative support for evaluating the development process. Depending on the grain of the analysis, there are several QA decisions that can be supported: the common underlying basis is to get information from *process measurements*. Depending on what we are able to measure, the supported QA decisions may regard: the improvement of the most critical phases of the development process, improvement actions on the most critical components, actions against the most critical suppliers, and/or the most critical actors (e.g., testing teams) involved in the process. Which specific action to take depends on data analysed, on gained insights (i.e., critical paths), and on the company policies. In the following, we report examples of (statistical) analyses that we plan to conduct on the case studies identified during the ICEBERG project.

Description

The practice of software defect analysis is recognized as an essential task for software process measurement. However, its effective application in industry raises several challenges. Data about defects experienced during the software lifecycle are a valuable source of information for product and process quality assessment and improvement. Defect tracking and analysis is therefore a practice recommended by the most important software process standards. There exist several methods using defect data attributes (such as the type, the trigger, the injection/detection phase, the impact) for tracking the quality of development artefacts and of process activities, so as to reveal inefficiencies and support process improvement. Two successful methods are the Orthogonal Defect Classification (ODC) [23], and the HP classification [24], both conceived to categorize the defects observed and then relate their occurrence pattern to process phases/activities.

Despite the success of such methods, implementing defect analysis into the industrial practice is heavily conditioned by the context, which finally dictates the objectives of the analysis and the constraints as well, significantly restricting the choice. There is, in fact, a trade-off between the target of the analysis, its potential outcomes, its extensiveness to several process aspects, and the cost required to implement it. For instance, analysing the process through software reliability growth models (SRGMs) are easy to implement, because they only require tracking the defect detection time as input, with not much personnel involved, and few changes to the process; however, they provide limited insights into the process, referred to the trend of the testing stage treated as a black box. Oppositely, implementing schemes as ODC or HP classification yields much information on single phases efficiency, but at relatively higher expense. Their application into real industrial settings can be difficult [25], [26], because of start-up costs (e.g., training, process changes), of required customizations (e.g., [27]), of non-immediate visible gain, and of reluctance of people to change their routine job.

In the ICEBERG project, we are therefore developing a lightweight defect analysis approach, merging various types of analyses and models to both product and process evaluation, which requires a minimal set of defect information as input in order to keep the impact on current practices low. The objective is to evaluate the quality vs. effort balance of the development process so as to identify potential critical phases, components, or actors. We are going to implement a black-box approach, in which the evaluation is inferred from the available data, not requiring any process change or any additional effort to developers (e.g., to re-classify defects according to a predefined scheme as could be the case with ODC, HP schemes). This avoids expensive training, terminology alignment, imposition to suppliers, and other adaptation activities. The minimal requirement must be the usage of a defects tracking tool.

Specifically, the objectives of the defect analysis approach in relation to the iron's triangle factors of cost, quality, and time, are:

- Measuring the *efficacy* and the *efficiency* of the requirements/design/implementation processes (i.e., the processes of "building" the product). With efficacy we mean the ***quality*** – i.e., (un-)defectiveness - of what is produced, while with efficiency we mean the ***quality with respect to the effort*** (i.e., cost) to produce it.
- Measuring the *efficacy* and *efficiency* of the testing process. The task of tester is to expose failures, i.e., the defects manifestation. Hence, testing efficacy is the level of defectiveness exposed by testers, and efficiency relates this to the effort required.
- Measuring the *efficacy*, the *efficiency*, and the *internal quality* of the fixing process. These are intended, respectively, as the number of defects, the average fixing time, and the properties of the fixing process as a whole, taken as indirect guarantee of a correct fixing.

To this aim, the approach is developed in the three main steps depicted in Figure 4.

Figure 4: The key steps of the defect analysis approach

Measurement is the starting point of the approach. From gathered data, several analyses are enabled with either analytical or empirical models, or hybrid approaches (i.e., analytical/measurements-based). Examples of evaluations supported by data on which we are working are:

- Process phases evaluation (implementation, testing, debugging/fixing);
- Component quality assessment (e.g., in terms of reliability level) for quality bottlenecks identification;
- Fine-grain defects analysis with respect to Severity/Priority/Reproducibility attributes, providing feedback on tester/debuggers behaviour;
- Suppliers Evaluation in outsourcing-based developments.

This type of information corresponds to estimates of cost due to poor quality as caused by critical development issues: e.g., the time in which debuggers fix defects is a cost; the inefficiency of testing is a cost; the wrong management of priority is a cost; the high defectiveness of a component implemented by suppliers is a cost. All this information is therefore going to be measured by a set of *metrics* defined in the step 1 of Figure 4 as measurements are taken. In the following we report the list of analyses that we include in our approach. The flexibility of the method allows us to select the best set of analyses depending on data provided as input. The list also reports the output provided by the analysis and how these outputs are given to managers in terms of "metrics".

Input

The inputs required to implement the defect analysis approach for quality decision support are the ones typically collected in a bug-tracking tool. Depending on the details tracked about the defects, several analyses can be carried out.

The minimum requirement is the *Date and time* of the defect (or, more generically, issue) detection and effort measures (e.g., man-months for implementation, and man-month for testing).

Optionally, the method can take: the defect Priority, Severity (impact), Detection Phase (i.e., Design Review, Code Review, Unit Testing, Integration testing ,…), the defect type (according to some classification, such as IBM ODC, HP), the age of the code module (e.g., new, base, rewritten, re-fixed), the defect Trigger, the Source (in-house, outsourced, library, …), the reproducibility (e.g., always or not always reproducible).

These input parameters can be used for: derive quality vs. effort indicators, and identify problems and criticalities in the lifecycle (e.g., phase/activity/team causing low index value).

Table 1 summarizes the potential model's input. This is a superset, meaning that different analyses can be done depending on the input information.

Source	Data Type	Data
Bug Repository	Defect Data	*Severity/Reproducibility/Priority, Defect Triggering (and/or activity that made the defect surface, e.g., code review, inspection, unit testing, workload/stress testing, concurrency testing, operational usage), Defect Detection Phase, Supposed Defect Injection Phase, Fixing time, Defect fixing Phase, Defect Type, Defect Impact, Defect mode (wrong, missing), defect source, source age, work/Rework*
Source Code Repository	Code/Process Metrics	*Size Measures (LoC, #Req, Function Points), Complexity metrics (McCabe, Halstead's), Source File metrics, code churn/change metrics, version*
Development/Test Engineer	Effort Estimate	*Testing effort (e.g., man-months dedicated to testing)*
		T Maximum threshold given to the delivery time of the system.

Table 1: Model's Input

Output:

Table 2 summarizes the main decisions supported by this class of models synthetically, which is again a superset with respect to the usage that can be done of the input information. Note that some of the specified analyses are also detailed in the subsequent sections, being this defect analysis model at higher level.

With a greater detail, Table 3 summarizes the analyses that can be done by joining more input information pieces, and their output depending on the information recorded by the tester and/or the person in charge of fixing a defect (with minimum requirement

being only the detection time and date with effort measures). The analysis that we will carry out will depend on the availability of such information in the case studies. The analysis are intended as "statistical" analysis, with output always accompanied by a "confidence level" indication (e.g., a given metric value is greater than another, *with 95% of confidence*).

Decisions	Description
Release Policy	Quality (reliability) analysis/assessment and time to get a given quality
Testing decisions: how much effort to invest	From the analysis of the testing process (test efficacy, efficiency) and of the product quality (detected/expected defects) with respect to the effort devoted so far, decide on investing more or less resources
Testing decisions: if and how to change the current process based on defect data	Analysis of defects per severity/reproducibility/priority, of detection/injection phase, of defect triggering phase and activity, defect type, in order to identify mismatch (expected vs actual patterns)
Testing effort allocation	Prediction of defective modules from code/process metrics
Decision on Debugging Process improvement and Development improvement	Analysis of the bug fixing time, defect type, defect impact, defect source, defect source age, prediction of defective modules from code/process metrics to focus design efforts, analysis of defect features to get feedback on implementation

Table 2: Model's Output

Input Info	Joined with:	Type of Analysis	Output Info
On detection, tester will record:			
Opening Time		Reliability Analysis	Estimate of Expected Defects, Estimate of (expected) Reliability (i.e., non-failure probability), Estimate of Residual Defects. Both during testing and during operational phase
		Release Policy Analysis	Decisions on "When to stop testing, when to release", "What is the quality, under the current testing process, expected at the end of testing"
	Size measures: *LoC, #Req, Function Points*	"Normalized" reliability analysis	Estimated Expected Defects *Density*, Estimated Expected Residual Defects *Density*
	Effort measures: *testing effort (e.g., man-months)*	Test Efficacy and Efficiency Analysis	Test maturity (%): detected defects so far over the total expected defects, Test Efficiency: defect detection rate, Test Efficiency: percentage detection efficiency (progress in terms of "test maturity increase" per effort unit), Test Efficiency: relative efficiency in terms of

			"effort units (e.g., man-weeks) required to achieve a maturity of x%"
	severity/ reproducibility	severity/ reproducibility analysis; Cross-analysis with the previous ones	Defects per category: "which implementation has higher severe defects in the average? what is the trend of high-severe defects per implementation item? Do testers of different implementation use the same criteria to assign severity? Which testing activity exposes the most severe defects? Which percentage of "not-always reproducible" defects is found during testing and which percentage during operation (high-cost defects)? What testing activity exposes the "not-always" reproducible defects?
Defect Triggering (and/or activity)		V&V Analysis	Identification of critical phases of testing (e.g., function review, code review, testing) and operational conditions in which defects are found (during testing or at runtime); Identification of critical environmental conditions (e.g., high workload-stress greatly contributing to expose defects); "Signature" of testing techniques with respect to defects they are able to find (how many, of what type, of what impact in terms of severity)
Defect Detection Phase		V&V (Phase) Analysis	Identification of critical phases of testing - analysis of expected detection phase vs. actual detection phase; "Delay" and cost analysis of testing - thus cost analysis referred to defects that should have been detected earlier
Supposed Defect Injection Phase		Development and V&V Analysis; Defect Flow Analysis	Development Phase Analysis - which phase introduces more defects (and of what type, impact); Defect flow analysis: analysis of the latency (and cost) required to detect defects (for how many phases the defect flows and survives); analysis of V&V activities vs. latency
On fixing, debugger will record:			
Fixing time		Fixing process (debug) analysis	Efficacy: percentage of closed (or pending) defects; Efficiency: mean time to fix
		Fixing process evolution over time	Efficacy and Efficiency over time; Continuity of the process over time; homogeneity of the process (e.g., peakedness and skew of the fixing time distribution)
	severity/ priority/ reproducibility	Fine-grained Fixing process analysis (analyse potential causes for	Previous metrics normalized per average severity (have more severe defects required more time to be fixed)?; priority analysis (have defects at higher priority been fixed earlier?) ; reproducibility: have "not-always reproducible" been actually

		experienced time to fix)	more difficult to fix (thus justifying higher Time to fix)?
Actual working Time		Detailed Fixing process (debug) analysis; Latency Analysis	Analysis of the bug tracking tool usage (it is expected a small difference between actual and recorded time to fix); Latency analysis: when the actual fixing work starts with respect to the claimed time; percentage of actual time over recorded time
Defect fixing Phase		Detailed Fixing process (debug) analysis	When the defect has been fixed w.r.t. when it was to expected to be fixed (cost analysis like "detection vs. injection" analysis: in this case it is "correction vs. detection")
Defect Type		Development Analysis	"Signature" of defect types over the development phases: expected vs. experienced defect. Analysis of patterns of defect types vs. development phases in which they have been injected. Cross-analysis with many previous and following attributes: defect type vs. trigger, vs. V&V activities, vs. impact, vs. source , vs. age, vs. target; type-based defect prediction (see below)
Defect Impact		Development and V&V Impact Analysis	Crossed analysis with: development phases, V&V phases and activities, defect type and triggers, and others…
Defect Mode (missing, wrong)		Detailed Development and V&V Analysis	As above, differentiated per "missing" defects and "wrong" defects; feedback to developers
Source (in-house, outsourced, library)		"Source Defect" Analysis	How many defects per source item type (in-house, outsources); crossed analysis with previous attributes
Source Age (new, base, rewritten, refixed)		"Source Age" Analysis	Age is intended the age of the code affected by the defect as development history: base code from the previous release, new code from the current release, rewritten code or refixed code. This allows analysing the impact of reusing code, of regression bugs, of writing completely new code, of using a baseline. Crossed analysis with previous attributes makes sense also.
Target of the fix (e.g., source file)		Code-defect Relationship Analysis	How many defect (density) per target; how target (metrics) are related to defectiveness
Version		Defect Pattern Evolution across versions; release policy analysis	How defects (type, trigger, impact, age,…) evolves across versions; how releases relate to defects found in operation; how releases are related to fixing (e.g., release train effect)
Work-rework		Regression Likelihood Analysis	How many defects are opened during a re-work; likelihood of introducing regression

			bugs; crossed analysis with triggers (environmental conditions in which defects surface)
More advanced analysis. For internal quality and prediction			
Size and complexity metrics; CVS metrics (code churns, etc.)		Code-defects Relationship; Defect Prediction	Empirical models to build predictors of defectiveness in modules; can be customized per defect type
Requirements-, design-, organizational metrics		Process metrics-defects Relationship; Defect Prediction; Detailed phase analysis (relation between phases metrics and defects)	How metrics at each level are related to defects; this can be specialized per phase (e.g.,: how requirements metrics are related to, and can predict, defects of a given type, or defects injected in requirements phase, …)
Description of the defect; notes; discussions; number of state changes in the report, …		Communication; Topic analysis, semantic analysis	Relating communication patterns (length of discussion, topics inside, number of participants to the discussion) with time to fix
Test Effort per component		Optimal test effort allocation	Allocate effort to projects with higher expected defectiveness

Table 3: Input-Output matrix describing the Possible Analyses and output in relation to provided information

6.1.1 DEFECT PREDICTION

As remarked in the deliverable D2.2, defect prediction approaches basically provide guidelines to predict defects in source code by exploiting the usefulness of elementary metrics or previous defects. They have the following common steps that can be iterative and overlapping.

- ***Step 1.*** The metrics evaluation is accomplished. Depending on the adopted type of metrics (e.g., object-oriented metrics or "traditional" product metrics, like number of lines of code, McCabe complexity), different computing approaches are used.

- ***Step 2.*** The relationships between the values of the metrics and the numbers of bugs found in the system (e.g., in the classes) are discovered. Well-known statistical methods (e.g., logistic and linear regression) have been largely adopted to validate the usefulness of the metrics to identify defective classes. Basili et *al*. in [28], for example, validate object-oriented design metrics as quality indicators by using logistic regression technique [29]. In the contrast, Gyimóthy at *al*. in [30], besides using regression methods (logistic and linear regression), also employed machine learning techniques to validate the usefulness of object-oriented metrics for fault-proneness prediction on open source software

In order to validate the metrics' usefulness for fault-proneness, the output of the previous step is analyzed. Specifically, the values obtained are checked against the number of bugs found in the system (e.g., in [30] the values of the object-oriented metrics of the open source Web and e-mail suite called Mozilla are checked against the number of bugs found in its bug database called Bugzilla).[7]

Input

Defect prediction approaches utilize *software metrics* and *defect data* collected during the software development process. Their efficacy is, therefore, influenced by the relevance between software metrics and fault data. The modules predicted to be fault-prone will receive more inspection and testing, thereby improving their quality. The literature contains a wealth of software metrics proposed for software fault prediction. In fact, software metrics may be used in prediction models to improve software quality by predicting fault location [31]. In the deliverable D2.2 more details on the survey [31] can be found. However, we can remark that, in general, software metrics are categorized as follows:

- *Traditional:* size (e.g. LOC) and complexity metrics (e.g. McCabe [32]).

- *Object-oriented*: coupling, cohesion and inheritance source code metrics used at a class-level (e.g. Chidamber and Kemerer [33]).

[7] http://www.bugzilla.org/

- *Process:* process, code delta, code churn, history and developer metrics. These metrics are usually extracted from the combination of source code and repository, and they require more than one version of a software item.

Different prediction approaches have been introduced by relying on diverse information (e.g., on source code metrics, process metrics or previous defects). The efficacy of defect prediction models is influenced by relevance between software metrics and fault data [34].

Output

The *accuracy* and the *granularity* are two important qualities of software fault prediction algorithms [35]. The accuracy represents the degree to which the algorithm correctly identifies future faults. On the contrary, the granularity specifies the locality of the prediction. As remarked in [35], typical fault prediction granularities are: (i) the executable binary [36]; (ii) a module (often a directory of source code) [37]; (iii) or a source code file [38]. A directory level of granularity, for example, means that predictions indicate a fault will occur somewhere within a directory of source code. As stated in [35], the most difficult granularity for prediction is the entity level (or below), where an "entity" is a function or method.

Supported QA decisions: These models provide support for decisions both at design time and testing time. The modules predicted to be fault-prone will receive more inspection and testing, thereby improving their quality.

Example of Defect Prediction Approach One of the most known defect prediction approach is the Basili et. al approach [28]. Basili et *al*. have used eight projects developed by using a sequential life cycle model, a well-known OO analysis/design method. The projects were written by students in C/C++. Basili et *al.* have slightly adjusted some of CK metrics in order to reflect the specificities of C++. Based on empirical and quantitative analysis, they have argued that several of CK metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Moreover, they have also figured out that, on their data set, CK metrics are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes. *GEN++* [39] was used to extract CK metrics directly from the source code of the projects delivered at the end of the implementation phase.

6.2 BUILD-OR-BUY DECISIONS MODELS

In this section we devise the formulation of a cost minimization problem under reliability and delivery time constraints. This model stems from our previous work in the context of component-based software [40], where we have introduced a model to support build-or-buy decisions about software components while minimizing costs under quality constraints. Components can be either bought as COTS (commercial-off-the-shelf) products, and probably adapted to work in the new software system, or they can be developed in-house. We adopt a general definition of software component: it is a self-contained deployable software module containing data and operations, which

provides/requires services to/from other elementary elements. A component instance is a specific implementation of a component.

The model considers the following architectural decisions: (i) replacing existing software units with functionally equivalent instances available on the market, and (ii) replacing existing software components with functionally equivalent software components developed in-house. Therefore, we show how an optimization framework can support the decision whether to buy software components or to build them in-house upon designing a software architecture.

Description

Let S be a software architecture made of n software components. Let C_i be the i-th software component ($1 \leq i \leq n$). Let C_{ij} be the j-th instance of the i-th component, and with j =0 we represent the in-house developed instance.
Let us assume to be committed to assemble the system by the time T while ensuring a minimum reliability level R and spending a minimum amount of money.

Symbol	Description
R	Minimum threshold given to the reliability on demand of the system.
T	Maximum threshold given to the delivery time of the system.
n	Number of existing software component
m	Maximum number of COTS instances available for each component
c_{ij}	Cost of the instance C_{ij}
d_{ij}	Delivery time of the instance C_{ij}
s_i	Average number of invocations of C_i
μ_{ij}	Probability of failure on demand of the instance C_{ij}
c_i	Unitary development cost of the instance C_{i0}
t_i	Estimated development time of the instance C_{i0}
τ_i	Average time required to perform a test case of the instance C_{i0}
π_i	Testability of the instance C_{i0}
y_i	The instance in-house C_{i0} is selected
x_{ij}	The instance C_{ij} is selected
N^{tot}_i	Total number of tests performed on the in-house developed instance C_{i0}

Table 4: Model's parameters and variables

Table 4 summarizes the parameters and the variables of the model. The formulation of the optimization model is given by:

$$\min \quad \sum_{i=1}^{n} \left(c_i (t_i + \tau_i N_i^{tot}) y_i \right) + \sum_{j=1}^{m} c_{ij} x_{ij}$$

$$\max_{i=1,\ldots,n} \left(y_i (t_i + \tau_i N_i^{tot}) + \sum_{j=1}^{m} d_{ij} x_{ij} \right) \leq T$$

$$\prod_{i=1}^{n} e^{-((1-\rho_i) s_i y_i + \sum_{j=1}^{m} \mu_{ij} s_i x_{ij})} \geq R$$

$$\rho_i = \frac{1 - \pi_i}{(1 - \pi_i) + \pi_i (1 - \pi_i)^{(1-\pi_i) N_i^{tot}}}, \forall i = 1,\ldots, n$$

$$y_i + \sum_{j=1}^{m} x_{ij} = 1, \forall i = 1, \ldots, n$$

$$y_i \in \{0,1\}, \forall i = 1,\ldots, n$$

$$x_{ij} \in \{0,1\}, \forall i = 1,\ldots, n, \forall j = 1,\ldots, m$$

Input

A primary input to the model is represented by an UML-based architectural model composed of: (i) a Component Diagram describing software components and their interconnections, (ii) a set of Sequence Diagrams describing the possible execution scenarios. Model parameters include, for example: (1) quality attribute values (e.g., probability of failure on demand) of elementary software units; and (2) unitary cost to develop a software unit in-house. Details on the parameters estimation can be found in [40].

Output

The results of the optimization model consist of a set of architectural decisions. The solution of the optimization model indicates the instance to choose for each component (either one of the available software unit instance or an in-house developed one) in order to minimize the software costs under quality constraints. The model solution also indicates the amount of testing to be performed on each in-house instance in order to achieve a reliability level that allows the whole system to satisfy the reliability constraint.

In fact, the formulation of the optimization model involves further variables representing the amount of unit testing to be performed on each in-house developed software component. Since these variables enter the software cost and the software reliability formulation, the model can be used to determine not only the best assembly of software units to be bought or built, but also the best amount of testing to be performed on each in-house developed unit to fulfill the constraints while minimizing development costs. Indeed, testing on in-house software components aims at increasing reliability estimation, whereas testing on instance of software unit available only aims at reliability estimation, because lack of source code does not allow to localize and

remove faults unless additional wrapper code is designed. In our approach, we do not entail wrapper code because it would bring additional time and cost hard to quantify. Model solutions are obtained by means of a commercial non-linear solver and are compared to those provided by a previous approach.

Model Summary. Table 5 summarizes the primary input of the model. A primary input to the model is represented by parameters related to: (i) the components C_i, which are estimated by using the architectural model (i.e,, the average number of invocations) or entered by the user (e.g., its number of COTS available), (ii) the COTS instances (e.g., the cost and the delivery time), and (iii) the in-house components (e.g., the unitary development cost). A model solution provides the optimal "build-or-buy" strategy for component selection, as well as the number of tests to be performed on each in-house developed component instance (as summarized in Table 6). The solution guarantees a system reliability over the threshold R, a system delivery time under the threshold T while minimizing the whole system cost.

Source	Data Type	Data
Architectural Models	Existing Components	s_i Average number of invocations of C_i
User	Model Constraints	R Minimum threshold given to the reliability on demand of the system.
		T Maximum threshold given to the delivery time of the system.
	Existing Components	n Number of existing software components
		m Maximum number of COTS instances available for each component
	COTS instances	c_{ij} Cost of the instance C_{ij}
		d_{ij} Delivery time of the instance C_{ij}
		μ_{ij} Probability of failure on demand of the instance C_{ij}
	In-house instance	c_i Unitary development cost of the instance C_{i0}
		t_i Estimated development time of the instance C_{i0}
		τ_i Average time required to perform a test case of the instance C_{i0}
		π_i Testability of the instance C_{i0}

Table 5: Model's Input

Architectural Decisions	Description
Build-or-buy decisions for each component C_i	(i) Replacing C_i with the COTS instance C_{ij} or (ii) Develop-in-house the component C_i.
Testing decisions for the in-house developed instances	The model suggests the amount of unit testing to be performed on each in-house developed software component.

Table 6: Model's Output

6.2.1 QUANTIFYING THE INFLUENCE OF FAILURE REPAIR/MITIGATION COSTS

As discussed in the deliverable D2.2, reliability and costs together have been considered in different contexts, for example to provide guidelines in (1) evaluating the effort spent to test the software, deal with the resource allocation during the test process or quantify the costs of service failure repair/mitigation actions, or (2) comparing the costs of defect-detection techniques. In the deliverable D2.2, we provide a quite extensive list of these approaches.

In our previous work [41], we have presented an approach for service selection taking into account costs and reliability requirements. In particular, we have defined a set of optimization models that allow quantifying the costs of service failure repair/mitigation actions aimed at keeping the whole system reliability over a given threshold.

The aim of our approach is to define SBSs obtained by a combination of both provided loosely-coupled services and in-house developed services, while satisfying costs and reliability requirements. These systems should be able to satisfy both costs and reliability constraints under the hypothesis that repair and mitigation actions can be undertaken to maintain the service's reliability over a certain threshold. To this end, we have defined a service selection approach based on the definition of a set of optimization models whose goal is to minimize the overall application cost while guaranteeing the required level of reliability. The high level view of the proposed approach is sketched in Figure 5.

The goal of our work has been to introduce a set of optimization models that allow quantifying the costs of service failure repair/mitigation actions aimed at keeping the whole SBS reliability over a certain threshold.

We assume that a service-based system made of n nominal services has to be assembled, and for each nominal service S_i, several alternative implementations are available, which can be split into: (i) an in-house service implementation, (ii) service implementations available for purchase by providers.

On the basis of our previous work [40] (discussed in Section 6.2), we have first introduced an optimization model, called *base model*, aimed at selecting either in-house built or provided services with the goal of minimizing the SBS cost while guaranteeing a certain level of reliability. Thereafter we have strengthened the reliability constraints, and we have built two different optimization models that aim to solve the same problem under new constraints, where one model starts from the solution obtained in the original model and tries to improve it (i.e., *base model with repair model)*, while the other one looks for an optimal solution in the whole search space (i.e., *robust model*). Finally, we have introduced a fourth model, based on stochastic optimization (i.e., *stochastic model*), with the goal of rather searching for solutions that explicitly take into account the stochastic nature of the problem and search for new repair/mitigation actions cheaper than the ones identified by the other models.

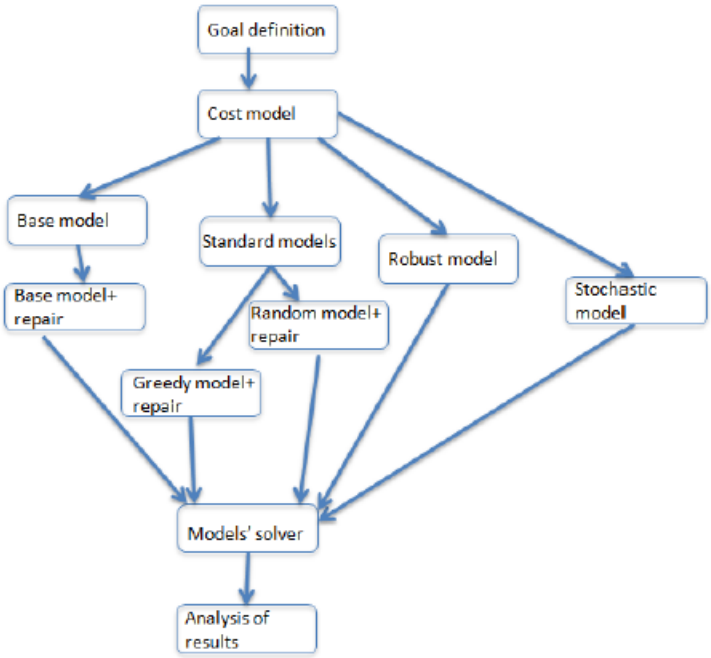

Figure 5: High-level Approach Overview [41]

Input

The input of our approach is the set of functional and non-functional requirements representing the goal/objective of the SBS to-be. A primary input of the models is represented by an UML-based architectural model composed of: (i) a Component Diagram describing software components and their interconnections, (ii) a set of Sequence Diagrams describing the possible execution scenarios.

Models' parameters include, for example: (1) quality attribute values (e.g., probability of failure on demand) of elementary software units; and (2) unitary cost to develop a software unit in-house.

As remarked above, the formulation of the *base model* stems from our previous work in the context of component-based software [40] (see Table 4 for the model's parameters and variables of this model). Specifically, with respect to the original model, here: (i) we have plugged the problem in a service-oriented paradigm, where the build-or-buy decisions refer to services rather than components, (ii) we have refined the software development cost function (that in the original work was a linear function of the development time) with a COCOMO II cost function [42], and (iii) we have removed the delivery time constraint, for sake of focusing on reliability concerns.

In particular, we have exploited the COCOMO II model [42] to define the development cost c_i of an in-house service. The COCOMO II model introduces a software cost function that depends on the *size* (i.e., the lines of code) and the *type* (i.e., simple, intermediate and complex) of software. Such attributes allow estimating the amount of effort, in terms of personmonths, needed to deliver the software.

We have adapted the COCOMO model by considering the amount of testing to be performed on an in-house software unit. In particular, we have introduced N_i, the number of tests performed on a service before delivery. The variables N_i appear both in the development cost function and in the reliability constraint.

We have introduced the following cost function for an in-house developed service:

$$c_i = cost_{pm} \cdot \left(\left((a \cdot size_i)^b \cdot (1 - testperc) \right) + pmt \cdot N_i \right) \quad (1)$$

where:

- $(a \cdot size_i)^b$ is the COCOMO II model for the development personmonths of a service by $size_i$, where constants *a* and *b* depend on the software size and type.
- $(1 - testperc)$ is the percentage of development effort that is not spent in testing.
- $pmt \cdot N_i$ is the effort spent in testing.
- $cost_{pm}$ is the cost of a personmonth.

Similarly to Table 5, Table 7 summarizes the primary input of the *basic model*.

Source	Data Type	Data
Architectural Models	Nominal Services	inv_i Average number of invocations of S_i across all considered interaction scenarios.
User	Model Constraints	*R* Minimum threshold given to the reliability on demand of the system.
	Nominal Services	*n* Number of nominal services
		m Maximum number of service implementations available for purchase by providers for each nominal service.
	Service implementations available for purchase	c_{ij} Cost of the instance *j*-th
		μ_{ij} Probability of failure on demand of the instance *j*-th.
	In-house instance	c_i Unitary development cost of in-house service (estimated with Equation (1))
		π_i Testability of the in-house instance

Table 7: Basic Model's Input

Similarly, to our previous model [40] (see Table 6), the *basic model* support "buil-or-buy" decisions, namely it selects either in-house built or provided services with the goal of minimizing the SBS cost while guaranteeing a certain level of reliability. The model also suggests the number of tests to be performed on each in-house developed service instance.

Output

The solution of the set of optimization models can give insights on the service composition that best fit the requirements considering an explicit cost model and the possibility to define repair actions to improve the system reliability. This approach can help software architects in the decision-making process of assembling architectures satisfying quality requirements.

6.3 OPTIMIZATION OF ADAPTATION PLANS WITH COST AND QUALITY TRADEOFF

As explained in the deliverable D2.2, research in software adaptation has seen a flourish in the past years, in particular in the fields of new formalisms, tools, techniques, and development methodologies.

Usually, the goal of existing approaches is to predict and/or analyze some quality attributes, like performance or reliability, starting from the architectural description of the system, or to select the architecture of the system among a finite set of candidates that better fulfill the required quality.

In our previous work [43], we have addressed the problem of system quality from a different point of view: starting from the description of the system and from a set of new requirements, we devise the set of actions to be accomplished to obtain a new architecture. This is able to both fulfill the new requirements with the minimum cost and guarantee given levels of reliability, availability and performance.

The goal of our optimization model is to provide a support to the decisions that software architects take for adapting a Service-Oriented Architecture (SOA).

The optimization model minimizes the adaptation costs of the system in correspondence with a certain *change scenario* (i.e., a set of new requirements), while guaranteeing required levels of reliability, availability and performance.

Since our model may support different service application domains, we have adopted a general definition of software service: it is a self-contained deployable software module containing data and operations, which provides/requires services to/from other elementary elements. A service instance is a specific implementation of a service.

Different kinds of adaptation decisions could be made depending on several factors due mainly to the particular adaptation phase where the model is adopted (e.g., if run-time modifications are claimed, then it is only required the substitution of an unsuitable service without using more sophisticated actions).

In this paper, we have considered the following adaptation actions:

- *Introducing new software services*: An adaptation action may suggest to embed into the system one or more new elementary software services.
- *Replacing existing service instances with functionally equivalent ones*: An adaptation action may suggest to replace a service with one of additional instances available for it.
- *Modifying the interactions among services in a certain external service*: An adaptation action may suggest to modify the system dynamics by

introducing/removing interactions among services within a certain external service.

Any combination of such adaptation actions may have a considerable impact on the cost, performance, reliability and availability of the SOA. Therefore, our optimization model aims to quantify such impact to suggest the best adaptation plan, which still minimizes the costs while satisfying the performance, reliability and availability constraints.

Description

Let *S* be a SOA composed by n elementary software services. Let s_i be the i-th existing elementary service $(1 \leq i \leq n)$. Through the composition of its elementary software services, the system offers services to users.

Let $Inst_i$ be the set of instances for s_i, while s_{ij} represents the j-th instance of $Inst_i$. Let *NewS* be the set of new available services that can provide different functionalities, whereas $news_h$ represent the h-th service of *NewS*.

Table 8 summarizes the main parameters and the variables of the model. Figure 6 reports the formulation of the optimization model.

Symbol	Description
n	Number of elementary software service
$Inst_i$	Set of alternative instances for s_i
$NewS$	Set of new available services
c_{ij}	Cost of the instance s_{ij}
$\bar{c}_h$	Cost of the service $news_h$
inv_{ki}	Average number of invocations of the existing service i
inv_{kh}	Average number of invocations of the new service h
$q_{ij}, q \in \{r, a\}$	r_{ij} (a_{ij}) is the reliability (availability) on demand of the instance s_{ij}
$\bar{q}_h, \bar{q} \in \{\bar{r}, \bar{a}\}$	$\bar{r}_h$ ($\bar{a}_h$) is the reliability (availability) on demand of the service $news_h$
$Q \in \{R, A\}$	*R (A)* Minimum threshold given to the reliability (availability) on demand of the system.
Res	Maximum threshold given to the system response time.
K	Number of services offered by the system
Λ_k	Probability that the service *k* will be invoked
rt_{ij}	Response time of the instance s_{ij}
$\bar{rt}_h$	Response time of the new service h
x_{ij}	The instance s_{ij} is selected
z_h	The service $news_h$ is selected
y_{rp}	The adaptation plan p is selected for the new requirement r

Table 8: Main model's parameters and variables

$$min \quad \sum_{i=1}^{n}\sum_{j=1}^{|Inst_i|} c_{ij}x_{ij} + \sum_{h=1}^{|NewS|} \overline{c_h} z_h$$

$$\log Q_k = \sum_{i=1}^{n} (inv_{ki} \cdot \sum_{j=1}^{|Inst_i|} x_{ij} \cdot \log q_{ij}) + \sum_{h=1}^{|NewS|} inv_{kh} \cdot \log \overline{q_h} \cdot z_h, \forall k = 1 \ldots K$$

$$\sum_{k=1}^{K} \Lambda_k \cdot Q_k \geq Q$$

$$Rt_k = \begin{cases} \max\limits_{l' \in d(w)} Rt_{l'k} & w \in \overline{F}_k \\ \sum\limits_{i=1, i \prec_{dd} w}^{n} inv'_{ki} \cdot \sum\limits_{j=1}^{|Inst_i|} x_{ij} rt_{ij} + \\ + \sum\limits_{h=1, h \prec_{dd} w}^{|NewS|} inv'_{kh} z_h \overline{rt}_h + \\ + \sum\limits_{f \in \overline{F}_k, f \prec_{dd} w} \overline{inv}_{kf} Rt_{fk} & w \notin \overline{F}_k \end{cases}$$

$$\sum_{k=1}^{K} \Lambda_k Rt_k \leq Res$$

$x_{ij} \in \{0, 1\}, \forall i = 1 \ldots n, \forall j = 1 \ldots |Inst_i|$

$z_h \in \{0, 1\}, \forall h = 1 \ldots |NewS|$

$y_{rp} \in \{0, 1\}, \forall r = 1 \ldots m, \forall p = 1 \ldots |AP_r|$

Other Constraints (e.g., (3) and (4))

Figure 6: Optimization Model Formulation [43]

Input

The input of our approach is a *change scenario*, namely a set of new requirements that induce changes in the structural and behavioral architecture of the software system. Specifically, in our model, we consider as possible changes the introduction of new functionalities and the modification of the dynamics of existing functionalities. For each new requirement in a *change scenario*, we consider the different sets of adaptation actions (called *adaptation plans*) able to guarantee this new requirement. In this way, we obtain a set of decisions that lead to the definition of a new architecture, which minimizes the costs while keeping the reliability, availability and the response time within certain thresholds.

Output

The model suggests a new system architecture. A new architecture is, thus, obtained by modifying both its structure and its behavior. Specifically, in order to modify the software structure, the model replaces existing software services with different available services and/or embeds new software services into the system. With respect to the changes in the system behavior, it modifies the system scenarios (represented, for example, as BPEL processes [44]) by removing or introducing interactions between existing services and/or between existing and new services.

Model Summary. Table 9 summarizes the primary input of the model. In particular, the input consist of the following data. (i) A *change scenario*, namely a set of new requirements that induce changes in the structural and behavioral architecture of the software system. (ii) For each new requirement, the model gets as input value adaptation plans. An adaptation plan is a set of actions that address the requirement. Adaptation plan may be defined by a *User* (or "system designer" or "system maintainer") and/or automatically by the system itself (in case of self-adaptation). (iii) For each elementary service s_i, the model predicts its average number of invocations (i.,e., inv_{ki},) in the k-th service offered by the system, after the application of the application plans. inv_{ki} is estimated by analyzing the system architectural model, and using data associated with the adaptation plans (see [43] for details). Similarly, for each new service $news_h$, the model predicts its average number of invocations (i.,e., inv_h) in the k-th service offered by the system, after the application of the application plans. (iv) Finally, the models get as input values, for example, the cost of the alternative instance s_{ij}, or the response time of the new service $news_h$.

Any combination of adaptation actions may have a considerable impact on the cost, reliability/availability and performance of the software architecture. Our optimization model aims to quantify such impact in order to suggest the best adaptation plan, which minimizes the costs while satisfying the reliability, availability and performance constraints. The model suggests a new software architecture (as summarized in Table 10). Specifically, in order to modify the software structure, the model suggests how to replace existing software services with different available services and/or embeds new software services into the system. With respect to the changes in the system behavior, the model modifies the system scenarios (represented, for example, as BPEL processes) by removing or introducing interactions between existing services and/or between existing and new services. The model's solution guarantees a system reliability (availability) over the threshold R (A), a system response time under the threshold Res while minimizing the whole system cost.

Source	Data Type	Description
User	Change Scenario	*A* set of new requirements that induce changes in the structural and behavioral architecture of the software system.
User or the System itself	Adaptation plan	Set of actions that address a certain requirement.
Architectural Models and User	Existing elementary software services	inv_{ki} Average number of invocations of the existing service *i*
	New elementary software services	inv_{kh} Average number of invocations of the new service *h*
User	Existing elementary software services	*n* Number of elementary software service
		$Inst_i$ Set of alternative instances for s_i
		c_{ij} Cost of the instance s_{ij}
		r_{ij} (a_{ij}) Reliability (availability) on demand of the instance s_{ij}
		rt_{ij} Response time of the instance s_{ij}
	New elementary software services	*NewS* Set of new available services
		$\bar{c}_h$ Cost of the service $news_h$
		$\bar{r}_h$ ($\bar{a}_h$) is the reliability (availability) on demand of the service $news_h$
		$\bar{rt}_h$ Response time of the new service *h*
		K Number of services offered by the system
		Λ_k Probability that the service *k* will be invoked
	Model Constraints	*R (A)* Minimum threshold given to the reliability (availability) on demand of the system.
		Res Maximum threshold given to the system response time.

Table 9: Model's Input

Architectural Decisions	Description
Software Structure	(i)Replacing existing software services with different available services, and/or (ii) Embedding new software services into the system.
Software Behavior	The system scenarios are modified by removing or introducing interactions between existing services and/or between existing and new services.

Table 10: Model's Output

7 SCHEDULE/TIME DECISION-MAKING MODELS

7.1 RELEASE PLANNING

Description

A common problem in software testing is to decide when it is the best time to release the product, or, conversely, what is the quality if one releases the product after a given amount of testing time.

Software Reliability Growth Models (SRGMs) are useful means to these aims. A SRGM is a model describing how reliability grows as software is improved during testing by faults detection and removal. These models are usually calibrated using failure data collected during testing, namely fitting inter-failure times, and observing the variation of the failure intensity (number of failures per time unit) with testing time. They are used to answer questions such as "how long to test a software", or "how many faults are likely to remain".

We consider the most common class of SRGMs, those that describe the failing process as a non-homogeneous Poisson process (NHPP). These are characterized by the parameter of the stochastic process, $\lambda(t)$, indicating the failure intensity, and by the mean value function (*mvf*), m(t), that is the expectation of the cumulative number of defects detected at time *t* [45]:

$N(t)$: $m(t) = E[N(t)]$; $dm(t)/dt = \lambda(t)$

The mean value function provides indication on how many defects are being detected over time, and how many defects are expected to be found at a certain testing time *t*. Different types of SRGMs can be described by their mean value function, that appears in this form: $m(t) = aF(t)$, where *a* is the expected number of total defects.

Many models have been proposed in the literature, and several tools have been developed to deal with parameterization and fitting of models (such as SMERFS, SoRel, PISRAT, and CASRE). For our purpose, we consider the list reported in Table 11 because of their wide spread in the literature and of their ability to capture several different potential behaviours of the testing process. In particular, we use the model proposed by Goel and Okumoto [46], which describes the failing process by an exponential *mvf* distribution, as it is one of the most successful and popular models for reliability growth analysis. The Delayed S-Shaped curve [47], also very popular, has been proposed in order to capture the possible increasing/decreasing behaviour of the failure rate during the testing process. With similar purposes, the logistic-based distributions (namely, the log-logistic [48] and the truncated logistic [49]) describe the processes in which the initial phase of testing is characterized by a slow increase because of the gradual improvement of testers skills in the initial learning phase, and because of defects being mutually dependent (i.e., some defects are not detectable before some others are). We also consider the generalized version of the Goel-Okumoto model capturing the S-Shaped nature of software failure occurrence, wherein Goel simply proposed an additional parameter turning the exponential into a Weibull distribution [50]. Finally, the normal-based (log- and truncated- normal) SRGMs are considered as they demonstrated a noticeable ability to fit a wide variety of reliability

growth scenarios and to software failure data collected in real software projects [51], [52].

Model	*m(t)* function
Exponential [46]	$a \cdot (1 - exp(-bt))$
S-shaped [47]	$a \cdot [1 - (1 + gt)exp(-bt)]$
Weibull [50]	$a \cdot (1 - exp(-bt^{\gamma}))$
Log Logistic [48]	$a \cdot (\lambda t)^{\kappa} / (1+(\lambda t)^{\kappa})$
Log Normal [51]	$a \cdot \Phi((log(t)-\mu)/ \sigma)$
Truncated Logistic [49]	$a \cdot (1-exp(-t/\kappa))/ (1+exp(-(t-\lambda)/\kappa))$
Truncated Normal [52]	$a \cdot (\Phi((t-\mu)/\sigma)) / (1-\Phi(-\mu/\sigma))$
* Φ *indicates the normal distribution*	

Table 11: Software Reliability Growth Models

For the approach we are developing within the ICEBERG project, we consider the list reported in Table 11 in order to capture several different potential behaviors of the testing process. It also reports the corresponding expression of the mean value function (*mvf*); the estimated number of defects is always the *mvf* first parameter, *a*. In the formulation of the approach, we considered that, in practice, there is no model to fit all the situations. Thus, our approach to use this type of models is to fit the testing data of a system/component with all the considered SRGMs, by using the EM algorithm [53], and then choose the best one. In particular, we perform a goodness of fit (GoF) test by means of the Kolmogorov-Smirnov (KS) test (with 90% confidence level), and compute the (adjusted) R-square coefficient to determine the goodness of fit of each model. We discard models with KS test satisfied or with an R-square less than 0.8. Among the remaining models, the best one is chosen by adopting the *Akaike Information Criterion* (AIC) method, i.e., choosing the SRGM with the lowest AIC value. Figure 7 shows an example of how these models are used for the best release schedule problem.

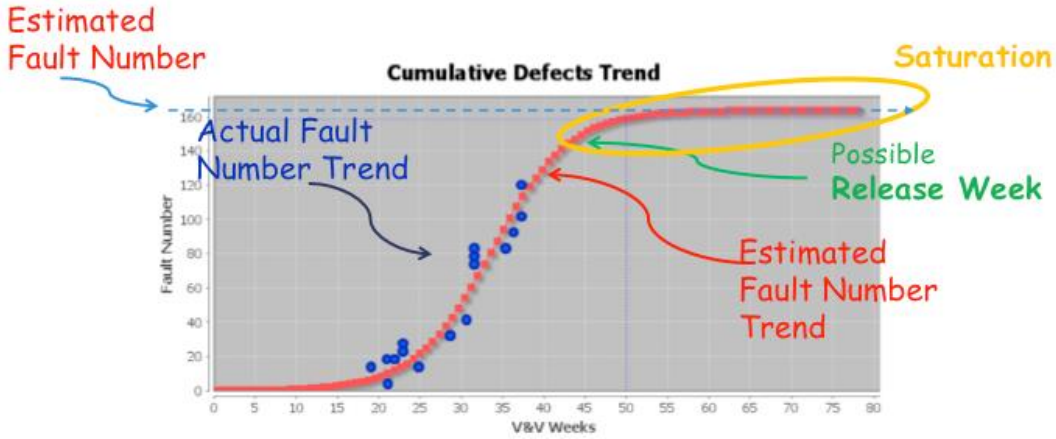

Figure 7: SRGM approach for release schedule determination

The SRGM best fitting these data is able to estimate the cumulative fault number trends, catching the eventual situation in which testing will no longer produce benefits (i.e., the "saturation"). This information is used to decide on the best time to release considering the achieved testing cost/product quality trade-off. Besides using the model for this type of decision (others are described in the "output" section), we use SRGM also in the models presented below to support other types of decisions.

Source	Data Type	Data
Bug Repository	Defect Data	*"Opening Time" of the issue describing the defect*

Table 12: Model's Input

Decisions	Description
Release Planning	Quality (reliability) analysis/assessment and time/effort still needed to attain a given quality, i.e., prediction of the optimal time to release, given a quality to achieve
Testing decisions: how much effort to invest to achieve a desired quality	From the analysis of the reliability growth decide on how much effort to devote yet., which actions to take to align predicted vs achieved quality

Table 13: Model's Output

Input:

This model requires, as input, the opening time of defects discovered during testing (and/or during operation), as summarized in Table 12. It is conceived to get a minimal amount of data in order to not overwhelm companies with additional activities to do (e.g., classifying defects), but just getting information needed in a non-invasive way.

Output and Supported SQ Decisions:

As output, the model will provide: *i)* estimates of the achieved software reliability at a given time, as well as of the number of residual defects in the software after a given testing (or operational) time; *ii)* prediction of the expected reliability given a budget (e.g., in terms of testing time or testing effort) to spend on the system/component; *iii)* prediction of the optimal time to release required to achieve a given reliability level, *iv)* indications to identify delays and their causes in the process by comparing actual and predicted release as well as actual *vs* predicted quality. These indications support the decisions on how much testing is still necessary for a given system/component, what is the best time to release, which actions to take to align the achieved quality level with the predicted one (Table 13).

7.2 DEBUGGING ANALYSIS FOR IMPROVED RELEASE PLANNING

Description

As seen, SRGMs can be used to support decisions on testing effort investment and release planning, as well as for quality assessment. However, SGRMs make a set of assumptions to meet the mentioned objectives, the most usual ones being: immediate debugging, perfect debugging, dependent inter-failure times, equal probability to find a failure across time units. In literature, a greater attention is being paid to the *immediate debugging* assumption, since its impact is more relevant than other factors in real projects. While some work introduces modelling approaches to include repair times [54] [55], several empirical studies make it evident that debugging is a complex process to model in real-world projects [56], [57], [58]. Indeed, there are many factors impacting the computation of the actual repair time and the regularity of the debugging process, including the type of defect, its priority or severity, and human factors (e.g., skills of people involved in the fixing process). These make such an assumption easy to be violated (especially as complexity and size of a software project increase), with repair times far from being immediate. In many cases, the debugging process might even become a bottleneck for project releases, and its impact cannot be neglected at all.

The impact of an irregular and variable debugging process hampers a correct modelling and influences the assessment of release quality estimates and of SRGM-based predictions. This can determine errors in taking decisions on when to stop testing, as well as in the estimate of the residual defectiveness.

Besides the previous usage of SRGMs, we are adopting, in the ICEBERG project, the SRGMs also to analyse the debugging process, and consequently "adjust" the SQ decisions on release planning and testing effort investment depending on debugging. Specifically, we have analysed 3,392 real-world issues of an industrial case study of the medical scenario (described in the deliverable D2.2) collected over two years. On these data, we first *i)* use SRGMs to characterize the software reliability growth under the assumption of immediate debugging; then, ii) we evaluate the impact of the debugging time evolution on reliability estimation and prediction, and thus on release scheduling performed by SRGMs. We show that:

1. Collected issues are amenable to be modelled by SRGMs; we applied a set of 7 models to fit data and found the truncated logistic and truncated normal SRGM being the best fitting models. Despite the real data do not fulfil classical SRGM assumptions, such as dependent inter-failure times and equal failure detection probability, the models have shown to be robust. However, since they are built upon opened issues, nothing can be said about the *non-immediate debugging* assumption. Therefore, these models are useful for predictions only provided that the debugging time is negligible.
2. The observed debugging process has a non-negligible time, on average equals to 12.8 days. The non-immediate debugging has an impact on both reliability estimation and prediction, and thus on optimal release time, in a different way. In both cases the impact is dependent on the debugging process quality in terms of debugging time and debugging time variation, but while the impact on reliability estimation is quite insensitive with respect to the testing time dimension, the release schedule prediction error can greatly vary: as testing

times proceeds, the optimal release schedule prediction can be affected considerably by the debugging time.

Input

Input data are the same as the release planning model, as this model is based again on SRGM (Table 14), augmented by data on closing time of the issues, being the model conceived to include the impact of debugging.

Source	Data Type	Data
Bug Repository	Defect Data	*"Opening Time" of the issue describing the defect*
		"Closing Time" of the issue describing the defect

Table 14: Model's Input

The defect data we are considering are available with the opening and closing time of the corresponding issue by the tester and the debugger, respectively.

The model uses SRGMs to figure out what is the impact of the debugging process on the release time prediction made by the tester. It means that if a tester plans a release based on an "immediate debugging" assumption, the committed error can be assessed.

Figure 8 and 9 show the analysis via SRGM constructed with the opening time of the issues of an example project. As explained before, we select a model among a set of SRGMs by considering goodness of fit measures (specifically the GoF test of Kolmogorov-Smirnov, and the Akaike Information Critetrion – AIC). The list of models is the one considered in Table 11.

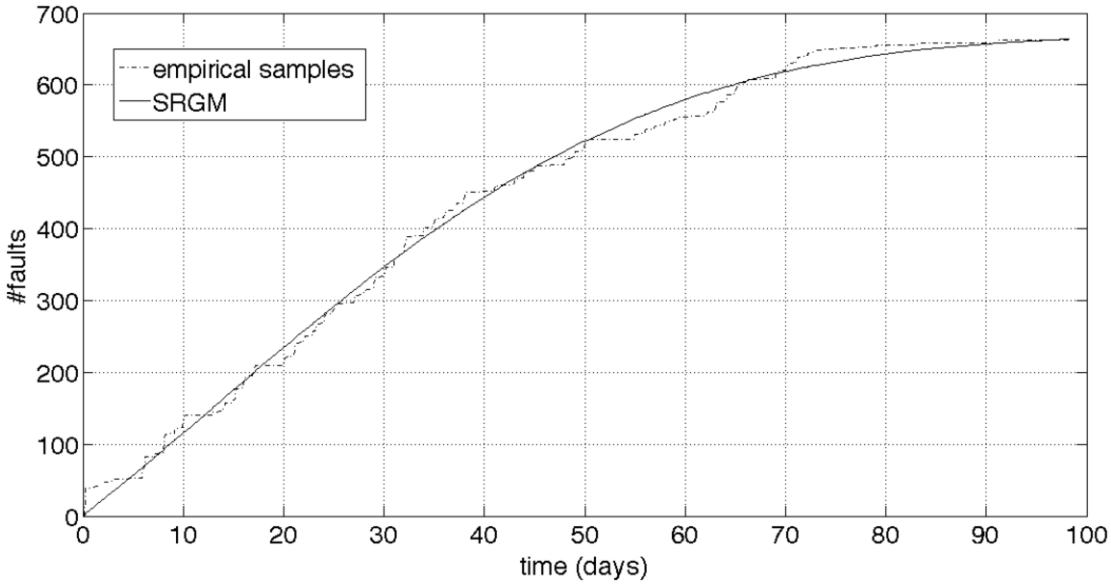

Figure 8: Empirical data and SRGM of the opening time of version 1

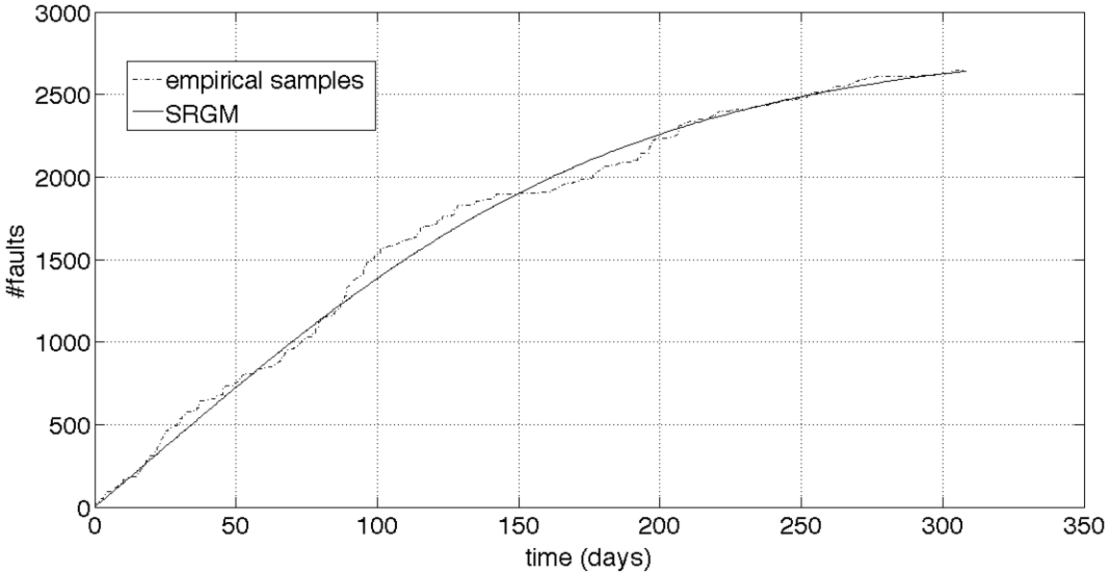

Figure 9: Empirical data and SRGM of the opening time of version 2

Version	*Current #Faults*	*Selected SRGM*	*Current Estimate of #Faults*	*KS Test True at*	*Exp. #Faults at t = ∞*	*Scale Param.*	*Shape Param.*	*AIC*
1	665	Trunc. Normal	663.93	90%	671.38	13.00	34.78	-1,491.7
2	2,647	Trunc. Logistic	2,640	90%	2,808.22	20.95	85.54	-6,834.9

Table 15: Statistics of the selected models

Table 15 shows the statistics of the selected models for the two versions. The estimates in this case are very close to actual data, both satisfying the KS test. Such models can provide estimates and predictions in terms of: residual faults at a given time, percentage of detected faults over the total expected ones; failure intensity; reliability. Note that these measures are equivalent to each other, since the expected cumulative number of faults at time *t* is the *mvf*(t) function, whose first derivative is the failure intensity function λ(t); the latter can be used in the computation of reliability. Considering these measures, testers can evaluate, for instance, what is the best time to release.

As for version 1, it is evident that the testing process saturates, detecting less and less faults as the testing proceeds. The process detected more than 99% of the total expected faults and will take much time to detect residual ones: thus this has been a good time to release. For version 2, testers detected roughly 94% of the total expected faults. If, for instance, they decide to release with the same quality as version 1, i.e., at 99%, the model predicts a testing time of 448 days, thus still 448 - 308 = 140 days of residual testing days. Based on these and similar analyses, decisions can be taken on when to stop testing.

The analysis has been conducted on the opening time of the issues. This means that 99% of quality is assumed to be the 99% of the total estimated faults that have been opened, i.e., detected: this is the actual released quality only under the assumption that

the correction of those faults was immediate. The actual quality is given by the closed issues, namely removed faults, whose fixing contributes to the actual reliability growth. In the next paragraph, we remove the immediate debugging assumption in the SRGMs, and discuss the changes in the reliability analysis.

Output

We consider, as output, the impact of debugging on the SRGM-based analysis. Figure 10 and 11 report the raw data about the cumulative number of opened (testing process) and closed (debugging process) issues, along with SRGMs fit- ting them. The graphs show what is the impact of the debugging times on the achieved quality.

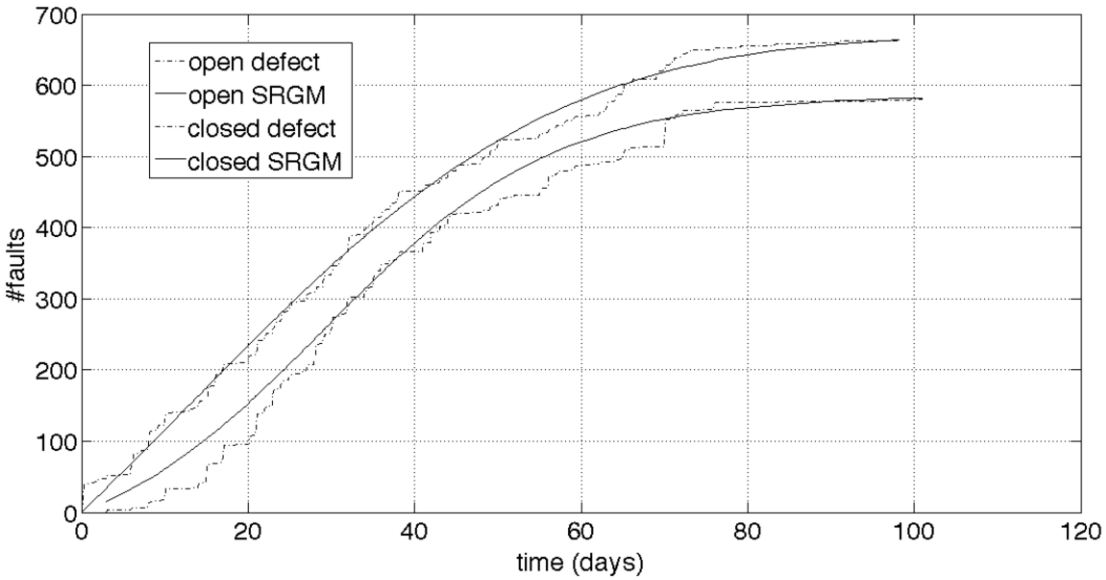

Figure 10:. Version 1 data

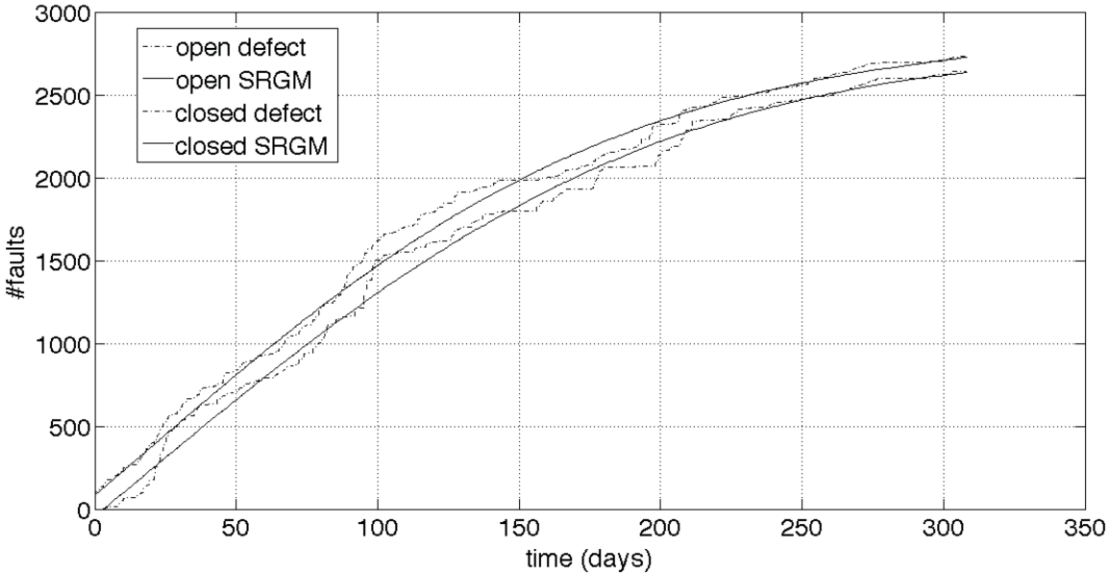

Figure 11: Version 2 data

The closed issue curve is the one actually contributing to reliability increase (namely, when the fault is actually removed); the opening curve would represent the reliability

increase only under immediate repairs. Thus, in the following, we consider the difference between the two curves and their corresponding models in order to infer conclusions about the debugging impact.

Let us define $\Delta_{issues(t)}$ and $\Delta_{time(F)}$, respectively, as: i) the difference between the opened and closed issues at time *t* (i.e., pending issues at *t*, which is the vertical distance between the raw data curves), and ii) the time required to close a given number of opened issues, F (namely, the delay of the debug process compared to testing, which is the horizontal distance between the raw data curves). We also define the differences between the corresponding models as $\rho_{mvf(t)}$ and $\rho_{time(F)}$. These are the differences between the opened and closed issues at time t and at $mvf = F$, respectively, as estimated/predicted by the corresponding SRGMs. The Δ values are used to: i) evaluate the difference between the actually achieved quality (in terms of number of removed faults) and the believed one[8], which is the quality under immediate repair assumption (i.e., the opened issues), as well as ii) the difference between the actual time required to close F issues and the believed time (again, under immediate debugging, through the opening curve). This is the impact of assuming immediate debugging on quality/time estimates. On the other hand, the ρ values are used to assess the same differences on predicted values, which are needed to take decisions like "when to stop testing". This is the impact of the immediate debugging assumption on predictions made through SRGMs.

For version 1, 99% of the total estimated issues has already been detected and it has been released, while the version 2 is still at 94% and it is still to be released: thus, we compute on version 1 the Δ differences on actual data to see the impact on estimated quality/time, whereas, on version 2, we compute the ρ values on future predictions, at percentages greater than the achieved 94%.

Let us first consider version 1. At the last day, 98.19, the total opened issues were 665, namely about the 99% of total estimated ones. The actual quality at that time is given by the closed issues, that are 578, thus the 86.14 % of the total estimated one, rather than the believed 99%. The error is therefore:

$$\epsilon_{\Delta issues} = \frac{\Delta issues(98.19)}{Closed(98.19)} \cdot 100 = \frac{665 - 578}{578} \cdot 100 = 15.05\%$$

where *Closed*(t) is the number of closed issues at time t. This means that if tester released actually at 99% of total issues and use the opening curve assuming immediate repair, the release quality is overestimated by 15.05 %.

Similarly, if tester used the opening curve assuming immediate repair, the removed 578 issues occurred, in his view, after 63.98 days, rather than at 98.19; thus there is a time estimation error of:

[8] Quality in the following is expressed through the (predicted) number of closed issues (i.e., removed faults) or the (predicted) percentage of closed issues with respect to the total one; for what said previously about the equivalence of this information to failure intensity and thus reliability, "quality estimation" and "quality prediction" are equivalent to "reliability estimation" and "reliability prediction".

$$\epsilon_{\Delta time} = \frac{\Delta time(578)}{ClosedTime(578)} \cdot 100 = \frac{98.19 - 63.98}{98.19} \cdot 100 = 34.84\%$$

where *ClosedTime*(F) is the time of closing of the *F*−th issue. This is interpreted as: using the immediate debugging assumption, the required quality is reached 34.84% later than the believed time. The first row of Table 16 reports results for release quality values from 95% to 98%, besides the mentioned 99% case.

Version	*Achieved or Predicted release quality*				
	95%	**96%**	**97%**	**98%**	**99%**
Version 1: $\epsilon_{\Delta issues}$	14.15%	14.59%	13.61%	14.06%	15.05%
Version 1: $\epsilon_{\Delta time}$	14.63%	16.07%	20.37%	31.28%	34.84%
Version 2: $\epsilon_{\rho mvf}$	0.04%	0.18%	0.33%	0.63%	1.05%
Version 2: $\epsilon_{\rho time}$	0.31%	1.18%	3.58%	14.89%	∞

Table 16: Results for release quality values

If the tester has not achieved a desired quality level yet, s/he may want to use SRGMs for a prediction and decide on when to stop testing. This is well represented by version 2. In this case, detected issues have been 2,647, namely the 94.4 % of total estimated ones. We evaluate the impact of debugging time on prediction accuracy supposing that tester wants to release at 95%, 96%, 97%, 98%, and 99% of total faults (namely: 2,662, 2,690, 2,718, 2,746, 2,774 faults). In these cases, if s/he uses the opening curve, the release should be at the days: 321, 339, 363, 396, and 447. However, using the closing curve, the number of removed faults corresponding to the above release days are: 2,661, 2,685, 2,709, 2,729, 2,745. Quality overestimation errors would be caused. For instance, suppose that tester wants to release at 97%. In this case, the quality overestimation error would be:

$$\epsilon_{\rho mvf} = \frac{\rho_{mvf}(363)}{SRGM(Closed(363))} \cdot 100 = \frac{2718 - 2709}{2709} \cdot 100 = 0.33\%$$

Similarly to the version 1 case, there will also be an error about the time prediction. If tester uses the opening curve assuming immediate debugging to release at 97%, the opened 2718 issues in 363 days will be closed only at day 376, causing an error of[9]:

[9] Note that, unlike the case of Δ_{time} value, here the difference is taken between the predicted time to close the number of issues that tester wants to remove and the predicted time to open that number of issues. For Δ_{time} values, we take the time to close the number of issues actually closed subtracted by the time at which that number of issues was opened (i.e., the "believed" time for achieving that quality).

$$\epsilon_{\rho time} = \frac{\rho_{time}(2718)}{SRGM(ClosedTime(2718))} \cdot 100 = \frac{376 - 363}{363} \cdot 100 = 3.58\%$$

where *SRGM(ClosedTime(F))* is the predicted time required to close F issues. The second part of Table 16 reports results form 95% to 99% release criteria.

As it may be noticed, the errors on quality overestimation are quite small in version 2, compared to version 1, and are slightly increasing with the desired quality level. The small error denotes a very good debugging process, whose curve is strictly following the opening one. Notwithstanding, it is interesting to note how the error on the time prediction is higher, and increases rapidly for increasing values of the desired release quality, due to the saturation of both curves. From 98% to 99%, it increases up to infinite. This is interpreted as follows: if tester wants to release at 98% of the total estimated faults, and uses the opening curve assuming immediate repair, it would predict a testing time of 14.89% days less than the actually required testing time. If this desired quality goes beyond the 98%, such an error increases abruptly, reaching infinite at 99%. Thus, depending on the desired quality and on debugging process characteristics, this testing time underestimation error may be very high and is much more sensitive than the quality overestimation error.

In general, such time error always goes to infinite at some point (precisely, at the saturation point of the closing curve); in the practice, it can go to infinite considerably earlier if the debug process is not as close to the testing process as in the version 2 case. For instance, in version 1, for the same type of error (computed on raw data) the infinite occur soon after 578 issues, i.e., at only 86.14 % of the total estimated ones.

To summarize, the worse the debugging process, the greater the error on quality estimation is, and the earlier the time prediction error goes to infinite: but while the quality estimation/prediction error is directly related to the number of pending issues quite independently from the release time (e.g., in the same way at 70%, 80%, or 90%), the time estimation/prediction error is much more sensitive: at high quality values, the underestimation of the required testing time can be very high, depending on the saturation of the opening and closing curves.

Supported SQ Decision

The non-immediate debugging has an impact on release and test planning decisions. With respect to the previous "release planning" model, this model adds the decision support with respect to the debugging process: the extent of the prediction error caused by a big average time to fix, or by a highly variable and irregular debugging time, can suggest engineers whether to invest on the improvement of the debugging process or not (and to what extent) with the aim of either reducing the mean fixing time and/or to reduce the variability (Table 17).

Decisions	Description
Release Planning	Quality (reliability) analysis/assessment and time/effort still needed to attain a given quality, i.e., prediction of the optimal time to release, given a quality to achieve
Testing decisions: how much effort to invest to achieve a desired quality	From the analysis of the reliability growth decide on how much effort to devote yet., which actions to take to align predicted vs achieved quality

Decision on debugging process improvement	Based on the prediction error that debugging causes, improve the mean time to fix a bug, and/or the reduce the variability

Table 17: Model's Output

8 SCHEDULE/TIME AND QUALITY DECISION-MAKING MODELS

8.1 RESOURCES ALLOCATION

Description

Testing accounts for a relevant part of the production cost of complex or critical software systems. Nevertheless, time and resources budgeted to testing are often underestimated with respect to the target quality goals. Test managers need engineering methods to perform appropriate choices in spending testing resources, so as to maximize the outcome.

The common industrial practice disregards such an important step. We believe that the main reasons are the lack of simple and tool supported methods, as well as the lack of evidence of success of the proposed approaches into real industrial contexts. Very generic criteria are typically applied in the practice, such as allocating resources driven by requirements (e.g., testing a component until all requirements have been tested at least once), or driven by the size (more testing to bigger modules). Sometimes, intuition drives testing choices: based on experience, a tester may deem one component more "critical" than another, therefore deserving more testing. As there may be relevant differences among components in terms of quality, their defectiveness can vary significantly. Moreover, a component can be newly developed, or it may be a reused unit that already underwent a functional testing phase, hence with a higher testing maturity. These differences intuitively call for a tailored engineering approach, in which more testing resources are spent where there is actually a greater need (i.e., poorer quality). Such an approach is expected to bring benefit in terms effort/quality trade-off.

In the project, we developed a model to decide dynamically how to allocate testing resources to software components, so as to minimize the estimated number of residual defects, and/or the estimated residual defect density, given a fixed testing budget. The method grounds upon software reliability growth models (SRGMs), used at component-level rather than at system-level as in the previous case, in order to monitor the testing progress of each component. From these, an estimate is obtained of the quality achievable for a component in relation to the testing effort devoted to it. Then, by iteratively solving an optimization problem, the next testing effort is directed towards the component that contributes the most to reduce the residual number of defects (density) in the overall system, thus improving the final trade-off between effort spent and residual defectiveness.

The test planning solution we have implemented is, unlike existing ones: *i)* dynamic, namely able of using testing data as they become available, exploiting them to adjust performance online, and robust with respect to variations during testing and volatility of planning time's assumptions; *ii)* simple in its application, and with as few assumptions as possible on the testing process; *iii)* ready-to-use, supported by an automatic tool.

Detailed Model description

Let us denote the expected number of residual defects as *E[Defects]*, and the expected residual defect density, measured in *#defects/KLoC*, as *E[Density]*. These are the two alternative objectives to minimize. For our purposes, components are autonomous, independently testable, and deployable units. The test manager has to distribute a budget B of testing resources (e.g., in number of man-weeks), among a set of

components; the *i*-th component will thus receive a testing effort equal to W_i man-weeks. The key idea is to use SRGMs to predict the detection ability of each component's testing process iteratively, and based on that, to allocate resources to components where testing will have the highest detection power. The method is based on the following main steps:

1. Initialization. Testing starts at time t0, when there may be no (previous) data available on testing of components to build any initial SRGM. Without any additional information, which could help to prioritize testing efforts at this stage, the initial allocation is done uniformly to all components, and the testing starts.

2. Start-up check. In this initial phase, at each time units (our time unit is the week), the method checks if the optimal allocation procedure can be applied with the available defect data. Specifically, we try to fit defect data of each component with every SRGM listed in Table 11 by the EM algorithm, and perform a goodness of fit (GoF) test by means of the Kolmogorov-Smirnov (KS) test (with 90% confidence level). If the test is satisfied for at least one SRGM, the component is ready for the subsequent step (it is said to be statistically valid) – in general we will have more SRGMs that fit one component, and will keep track of them for the next steps. This start-up check can be automatically repeated at each time unit from the beginning, or performed when the tester is confident that there are enough data for each component: in the practice, as rule of thumb, we observed that after no more than 20% of the total testing time there is at least one valid SRGM for every component. Thus, we advise to start checking from about 10-15% of the initially allotted testing time on.

As a guard, we conceived the possibility to skip to the next step also with only a subset of statistically valid components; in such a case (e.g., when there are components with very few and/or highly irregular data), the optimal allocation will apply only to that subset.

3. SRGM Selection. Given a number N of components with associated a set of statistically valid SRGMs, we select the best SRGM for each component by means of the *Akaike Information Criterion* (AIC) method, i.e., choosing the SRGM with the lowest AIC value. If, from the previous step, there is some component with no statistically valid SRGM, these are excluded from the optimal allocation strategy only for that iteration. These components will receive an amount of resources proportionally to their current detection rate.

4. Optimization. Depending on the goal (defect or density minimization), one of the following optimization problems is solved, using the *mvf* expressions of the SRGM selected for each component:

$$min! \qquad E[Defects] = \Sigma_i \, (EST_i - m(W_i^* + W_i))$$

$$s.t. \ \Sigma_i \ W_i \leq B^*$$

$$min! \qquad E[Defects] = \Sigma_i \, ((EST_i - m(W_i^* + W_i))/ \, \mathrm{SIZE}_i \,)$$

$$s.t.\ \Sigma_i\ W_i \leq B^*$$

where $i=1...N$, with N being the number of components; EST_i is the number of expected defects of the *i*-th component (the a parameter of its *mvf*); W_i is the testing effort to allocate to the *i*-th component; W_i^* is the testing effort already allocated to the *i*-th component; m(Wi*+Wi) is the (estimated) number of defects that would be removed if component *i* receives an effort of ($W_i^* + W_i$); $SIZE_i$ is the size of component *i* measured in *KLoC*, used to compute the defect density; B* is the residual budget at the current iteration [59].

5. Dynamic Update. W_i are the decision variables of the optimization problem, and are subject to the constraint that the total amount of allocated testing effort must not exceed the budget B*. This allows allocating efforts according to the prediction of the number of defects that will be found or of the defect density that will be achieved. However, as more data become available, the situation changes: it may happen that more data allow a more accurate estimation of residual defects (density), or, more importantly, the estimation can significantly deviate, because of changes in the testing process and thus in the detection trend. This calls for a dynamic approach, able to re-allocate resources from time to time, in order to "follow" the optimal solution, and exploit feedback coming from the testing process. Thus, after a predefined time (or when decided by the user), the defect data of each component are fitted again with every SRGM (by the KS test); step 3 (SRGM selection) and 4 (optimization problem) are taken again with the new data, starting a new iteration and re-allocating testing efforts accordingly.

The output are the testing efforts to allocate to each component at each time step as the testing proceeds.

Source	Data Type	Data
Architecture	Existing Components	Number and name of components
Bug Repository	Defect Data	"Opening time", for each component, of the issues corresponding to the defects

Table 18: Model's Input

Input:

For this model, the required inputs come from the bug tracking repository (Table 18), from which the opening times of defects that are detected during testing are used to build the SRGMs online. From these, given a testing budget (as further input) that mangers want to spend for testing, the allocation is performed dynamically, at any time the tester wants, by using the prediction of residual number of defects expected in each component.

Output and supported SQ Decision:

The provided output is the amount of effort (e.g., in man-months) to allocate to each system's components/modules in order to minimize the expected number of residual defects (Table 19). The model, if used in its dual form, can be used to estimate the number of residual defects after a given amount of testing time for each component.

Decisions	Description
Testing decisions: *Optimal test effort allocation*	Applying the model, decide how to distribute the effort available for testing to components, in order to minimize the total expected defectiveness

Table 19: Model's Output

8.2 PROBLEMS IN SOFTWARE PROJECT MANAGEMENT

As explained in the deliverable D2.2, SBSE techniques have also been largely applied in problems in software project management. A quite extensive list of these approaches can be found in [60].

As mentioned in [60], research efforts have been devoted for project scheduling and resource allocation. However, all these approaches basically provide guidelines to plan projects. Their primary input is represented by information about (i) work packages (e.g., cost, duration, dependencies), and (ii) staff skills. Shortly, as described in [60], they process these input information and produce the results, which consist of an optimal work package ordering and staff allocation. They are guided by a single or multi-objectives fitness function which it is typically minimized, for example: the completion time of the project, or the risks to associate to the development process (e.g., delays in the project completion time, or reduced budgets available).

An example of project management approach

The work in [61] defines a multiobjective optimisation technique based on genetic algorithms for simulation optimisation in order to help software project managers to find the best values for initial team size and time estimates for a given project so that cost, time and productivity are optimised.

Input:

Here below we list the input parameters that allow to model decision making regarding the initial team size and its composition, together with the initial estimations of project size and time to develop.

- *Initial Novice Workforce* (*NoviceWf*): The initial number of novice personnel allocated to the project.
- *Initial Experienced Workforce* (*ExpWf*): The initial number of experienced personnel allocated to the project.
- *Project Size* (*Size*): The estimate of project size.
- *Scheduled Time* (*SchldTime*): The estimate of project schedule.

Output:

The outputs of the model involve:

- *Project End* (*Time*): The final time of the project.
- *Cumulative Cost* (*Cost*): The final cost of the project.
- *Productivity* (*Prod*): The average productivity reached by the team through the project lifecycle. This is calculated as the ratio between size (*Function Points – FP)* and the *Project End* (time taken to finish the project).

This simple example would help choosing the best managerial balance between novice and expert personnel as well as schedule. Multiobjective approaches can provide a set of solutions that project managers can choose depending on the project.

Software Product Lines According to the SEI[10], Software Product Lines are a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

However, the products developed with this paradigm need to consider specific characteristics and combination of products. For this, SBSE techniques and models can also be applied to search for an optimal combination, testing, etc. this paradigm.

9 CONCLUSION

In this section, we present the overall conclusions of this document in the context of findings expected and novelty of our contribution.

To the best of our knowledge, this is the first attempt to combine existing SOTA solutions and SOPA practices. We will use two types of source: a) existing literature; and c) experience from the practice provided by experts in the field. Therefore, we envision that our quality-decision making frameworks will be helpful to enable practitioners to maximize the effectiveness of their specific software.

We believe that the adoption of well-assessed quality decision methods can be only be handled effectively by analysing effort and time necessary to incorporate them into real-world systems. Therefore, we plan to understand practitioners perceived strengths, limitations, and needs associated with using SOTA solutions in the industry. The outcomes will address, for example, the classical problem of: "How many tests are enough?"

General remarks in relation to our current view of SOPA are as follows: (i) the collection (and analysis) of few quality/cost/time data is in general performed, and (ii) practitioners consider quality, cost and time tradeoff important. In particular, the outcomes confirm our hypothesis that quality, cost, and time are highly relevant properties in next generation computing applications. Moreover, tradeoff analysis among multiple conflicting objectives should be supported, which is in general missing in the state of art and practice today.

We claim that addressing the highlighted challenges will require the contributions from researchers and industrial experts in different fields including not only optimization formulation (e.g., several metaheuristic techniques with different characteristics could be adopted depending on the nature of application domain), but also the integration of our frameworks in existing platforms.

[10] http://www.sei.cmu.edu/productlines/

10 REFERENCE

[1] Deliverable D2.1, “Industrial needs collection & state of the art surveys,” 7h Framework Programme IAPP Marie Curie program for project ICEBERG no. 324356, October 2013.

[2] Deliverable D2.2, "Validation scenarios and quality parameters," 7h Framework Programme IAPP Marie Curie program for project ICEBERG no. 324356, April 2014.

[3] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," In Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER '10, pages 411–414. ACM, 2010.

[4] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based Software Engineering: Trends, Techniques and Applications," ACM Comput. Surv., 45(1):11:1–11:61, Dec. 2012.

[5] V. Dhar and R. Stein, "Intelligent Decision Support Systems," Prentice Hall, 1997.

[6] D.J. Power "Decision support systems: concepts and resources for managers," Westport, Conn., Quorum Books, 2002.

[7] T. Xie et al., "Data mining for software engineering", IEEE Computer, 42(8), 55–62, 2009.

[8] M. Halkidi et al., "Data mining in software engineering," Intelligent Data Analysis 15, pp.413–44.

[9] A. Hassan, T. Xie, "Software Intelligence: the future of mining software engineering data" Future of Software Engineering Research, 161-165, 2010.

[10] Q. Taylor, C. Giraud-Carrier, "Application of data mining in software Engineering," Int. Journal Data Analysis Techniques and Strategies, 2(3), 243- 257, 2010 .

[11] J.J Dolado, "On the problem of the software cost function," Information and Software Technology, 43(1):61-72, 2001.

[12] M. Shepperd and C. Scho_eld. Estimating software project effort using analogies. Software Engineering, IEEE Transactions on, 23(11):736-743, nov 1997.

[13] J.S. Aguilar-Ruiz, J.C. Riquelme, I.Ramos, "Natural evolutionary coding: An application to estimating software development projects," In Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02), pages 1-8, New York, USA, 9-13 July 2002. AAAI.

[14] E. Alba, F. Chicano, "Software project management with gas," Information Sciences, 177(11):2380-2401, June 2007.

[15] A. Perini, A. Susi, and P. Avesani, "A machine learning approach to software requirements prioritization," IEEE Transactions on Software Engineering, Preprint (-):-, 2012.

[16] O. Räihä. "A survey on search-based software design," Computer Science Review, 4(4):203-249, November 2010.

[17] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, "A systematic literature review on fault prediction performance in software engineering," Transactions on Software Engineering, In Press-2011.

[18] C. Catal, and B. Diri. "A systematic review of software fault prediction studies. Expert Systems with Applications," 36(4):7346-7354, 2009.

[19] A. Shaukat, L.C. Briand, H. Hemmati, R.K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," IEEE Transactions on Software Engineering, 36(6):742-762, November-December 2010.

[20] B. Kitchenham, S.L. Peeger, B. McColl, S. Eagan, "An empirical study of maintenance and development estimation accuracy," Journal of Systems and Software, 64(1):57-77, 2002.

[21] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, "Automatically finding patches using genetic programming," In Proceedings of the 31st International Conference on Software Engineering (ICSE'09), ICSE'09, pages 364{374, Washington, DC, USA, 2009. IEEE Computer Society.

[22] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining software evolution to predict refactoring," In Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, pages 354-363, sept. 2007.

[23] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Wong, "Orthogonal defect classification – a concept for in-process measurements," IEEE Transactions on Software Engineering 1992; 18(11):943–956.

[24] R. Grady, "Practical Software Metrics For Project Management and Process Improvement," Hewlett-Packard Professional Books, 1992.

[25] N. Mellegard, M. Staron, F. Torner, "A light-weight defect classification scheme for embedded automotive software and its initial evaluation," Proc. 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE), 2012.

[26] S. Wagner, "Defect classification and defect types revisited," Proc. 2008 workshop on Defects in large software systems, 2008; 39–40.

[27] B. Freimut, C. Denger, M. Ketterer, "An industrial case study of implementing and validating defect classification for process improvement and quality management," Proc. 11th IEEE Int. Symposium on Software Metrics, 2005.

[28] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751–761, 1996.

[29] D. Hosmer and S. Lemeshow, "Applied Logistic Regression," Wiley-Interscience, 1989.

[30] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object oriented metrics on open source software for fault prediction," IEEE Trans. Software Eng., vol. 31, no. 10, pp. 897–910, 2005.

[31] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," Information & Software Technology, vol. 55, no. 8, pp. 1397–1418, 2013.

[32] T. McCabe, "A Complexity Measure," Software Engineering, IEEE Transactions on, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[33] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," Software Engineering, IEEE Transactions on, vol. 20, no. 6, pp. 476–493, Jun 1994.

[34] T. M. Khoshgoftaar, K.Gao, and A. Napolitano, "A Comparative Study of Different Strategies for Predicting Software Quality," in SEKE. Knowledge Systems Institute Graduate School, 2011, pp. 65–70.

[35] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in Proceedings of the 29th International Conference on Software Engineering, ser. ICSE '07. IEEE Computer Society, 2007, pp. 489–498.

[36] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, May 2005, pp. 284–292.

[37] A. Hassan and R. Holt, "The top ten list: dynamic fault prediction," in Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, Sept 2005, pp. 263–272.

[38] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," Software Engineering, IEEE Transactions on, vol. 31, no. 4, pp. 340–355, April 2005.

[39] P. T. Devanbu, "GENOA: A Customizable Language- and Front-end Independent Code Analyzer," in Proceedings of the 14th International Conference on Software Engineering, ser. ICSE '92. ACM, 1992, pp. 307–317.

[40] V. Cortellessa, I. Crnkovic, F.Marinelli, and P. Potena, "Experimenting the Automated Selection of COTS Components Based on Cost and System Requirements," Journal of Universal Computer Science, 14(8):1228–1255, 2008.

[41] V. Cortellessa, F. Marinelli, R. Mirandola, and P. Potena, "Quantifying the influence of failure repair/mitigation costs on service-based systems," in 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, IEEE, 2013.

[42] I. Sommerville, "Software engineering (9th ed.)," Addison Wesley, 2010.

[43] P. Potena, "Optimization of adaptation plans for a service-oriented architecture with cost, reliability, availability and performance tradeoff," Journal of Systems and Software, vol. 86, no. 3, pp. 624 – 648, 2013.

[44] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann,, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana, "Business Process Execution Language for Web Services," 2003 Version 1.1, May.

[45] S.S. Gokhale, K.S. Trivedi, "Log-logistic software reliability growth model," In: Proc. 3rd Int. High-Assurance Systems Engineering Symposium, pp. 34-41 (1998).

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

[47] S. Yamada, M. Ohba, S. Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection," IEEE Transactions on Reliability, R-32(5), 475-485 (1983).

[48] S.S. Gokhale, K.S. Trivedi,"Log-logistic software reliability growth model," In: Proc. 3rd

Int. High-Assurance Systems Engineering Symposium, pp. 34-41 (1998).

[49] T. Okamura, T. Dohi, S. Osaki, "EM algorithms for logistic software reliability models," In: Proc. 22nd IASTED Int. Conference on Software Engineering, pp. 263-268 (2004).

[50] A. L. Goel, "Software Reliability Models: Assumptions, Limitations and Applicability," IEEE Transactions on Software Engineering, SE-11(12), 1411-1423 (1985).

[51] H. Mullen, "The lognormal distribution of software failure rates: application to software reliability growth modeling," In: Proc. 9th Int. Symposium on Software Reliability Engineering (ISSRE), pp. 134-142 (1998).

[52] H. Okamura, T. Dohi, S. Osaki, "Software reliability growth model with normal distribution and its parameter estimation," In: Proc. Int. Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), pp. 411-416 (2011) R.E.

[53] H. Okamura, Y. Watanabe, T. Dohi, "An iterative scheme for maximum likelihood estimation in software reliability modeling," In: Proc. 14th Int. Symposium on Software Reliab. Eng. (ISSRE). IEEE CS Press, pp. 246256 (2003).

[54] J. D. Musa, A. Iannino, and K. Okumoto, "Software Reliability, Measurement, Prediction and Application," McGraw Hill (1987).

[55] C.-Y. Huang, W.-C. Huang, "Software Reliability Analysis and Measurement Using Finite and Infinite Server Queueing Models," IEEE Trans. on Reliability, 57 (1), pp. 192-203 (2008).

[56] T.T. Nguyen, T.N. Nguyen, E. Duesterwald, T. Klinger, P. Santhanam, "Inferring developer expertise through defect analysis," 34th International Conference on Software Engineering (ICSE), pp. 1297-1300 (2012).

[57] F. Zhang, F. Khomh, Y. Zou, A.E. Hassan, "An empirical study on factors impacting bug fixing time., 19th Working Conference on Reverse Engineering (WCRE), 2012.

[58] A. Ihara, M. Ohira, K. Matsumoto, "An analysis method for improving a bug modification process in open source software development," Proc. of the joint International Annual ERCIM workshops on Principles of software evolution (IWPSE) and Software evolution (Evol), pp. 135-144 (2009).

[59] G. Carrozza, R. Pietrantuono, S. Russo, "Dynamic test planning: a study in an industrial context," International Journal on Software Tools for Technology Transfer, 2014. pp. 1-15, Springer Berlin Heidelberg.

[60] F. Ferrucci, M. Harman, and F. Sarro, "Search-Based Software Project Management," Software Project Management in a Changing World (Springer), 2014, to appear.

[61] D. Rodriguez, M. Ruiz, J. C. Riquelme, and R. Harrison, "Multiobjective Simulation Optimisation in Software Project Management," in Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, ser. GECCO '11. ACM, 2011, pp. 1883–1890.