ICEBERG

ICEBERG models-based process

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.:	*3.3*
Deliverable Title:	**ICEBERG models-based process**
Organisation name of lead Contractor for this Deliverable:	*UAH*
Author(s):	*Pasqualina Potena, Luis Fernández*
Participant(s)	*Cigdem Gencel, Roberto Pietrantuono, Carmen Pagés*
Work package contributing to the deliverable:	*WP3*
Task contributing to the deliverable:	*3.1 and 3.2*
Total Number of Pages	*106*

Table of Versions

Version	Date	Version Description	Contributors
1.0	June 2015	First draft for review	Pasqualina Potena and Luis Fernandez
3.0	July 2015	Working draft for changes by DEISER and CINI	Pasqualina Potena and Luis Fernandez
4.0 and 5.0	September 2015	Reviewed by DEISER and CINI	Roberto Pietrantuono and Cigdem Gencel
6.0	October 2015	Final draft for formal approval	Pasqualina Potena, Luis Fernandez,Roberto Pietrantuono and Cigdem Gencel
7.0	3rd November 2015	Official final versión of deliverable 3.3	All partners
8.0	16th November 2015	Official final versión of deliverable 3.3 with small changes to guarantee consistency to deliverable 3.2 v2.	Pasqualina Potena, Luis Fernandez, Carmen Pagés, Roberto Pietrantuono and Cigdem Gencel: reviewed by all partners

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

The aim of D3.3 of the ICEBERG project "Model-based Process Definition" is to extend the deliverable D3.1 [1] by providing a more detailed presentation of the model-based decision making process and the generic framework, which have been under development in the ICEBERG project. In particular, we describe raw measurement/prediction models that would help in determining the cost of quality (and not-quality) and allow making best decisions for the trade-off between cost and quality, as well as a generic process definition for how to utilize such models in industrial settings.

2 INTRODUCTION

The goal of our work is to assist project managers and quality managers in making informed decisions during software development and maintenance. Informed decision-making requires collecting and analyzing quantitative data and providing the resultant information in an understandable way to decision makers.

Such assistance requires not only evaluating the dimensions of the well-known project management iron triangle, which are cost (e.g. cost to correct a bug during testing, or the cost of testing per unit testing-effort expenditures), time (e.g. time to detect and fix a bug), and quality (e.g. level of reliability), but also understanding the nature of interactions and tradeoffs among them to be able to make better decisions under different constraints.

In this document, first, we present the generic models-based decision making framework and process, which have been under development in the ICEBERG project during the last three work packages. Then, in the following chapters, we also provide three different instantiation of the models-based process defined for making various quality management decisions.

The following aspects characterize the novelty of each of these instantiation:

- **Optimal Allocation of Testing Resources.** We developed an automated optimization process for dynamically allocating testing resources to software modules (functionalities) based on trade-offs among software quality, cost, and schedule/time requirements. **We also explicitly consider uncertainty in the testing process in order to evaluate the robustness of the testing resource allocation.**

  In particular, our approach helps to: (i) select (and use) Software Reliability Growth Models (SRGMs) in order to make the software testing process more effective; and (ii) handle parameters uncertainty, which, as shown through our real world software project, plays a critical role in accurately describing a testing resource allocation process. It is well known that SRGMs sometimes show good performance in terms of predictability of the software reliability, but sometimes they do not. In this work, we show that the handling of uncertainty is a key factor for a trustworthy prediction of the reliability of a software system, and leads an optimization model to a more precise (and less pessimistic) estimation of the system reliability, as well as to a more effective and efficient testing resource allocation activity.

- **Optimal Regression Functional Testing.** Based on the generic models-based decision making process, we proposed an automated prioritization approach for large software systems that embeds the "code churn"

measure. Code churn represents a measure of the amount of code change taking place within a software system over time. Thus, we propose to use code-coverage measures (produced by static code analysis) by considering software system evolution metrics (extracted from system's change history).

- **Architectural Decision Making.** We developed an automated approach for making architectural decisions. Specifically, our focus is on (i) modeling and analysis of QoS tradeoffs of a software architecture based on optimization models, and (ii) definition of framework for supporting the software architects/maintainers. Thereby, we support software architects/maintainers to manage the interactions and conflicts between requirements, between design decisions, and between requirements and design decisions. The support includes automatic detection (by model checking techniques) of interactions and conflicts mostly in the part of the architecture design decisions and propagation of interaction between different levels. Our approach also allows producing the space of possible feasible architectural solutions obtained by instantiating parametric design decisions. Each solution is computed taken into account the specification constraints associated with the design decisions and the known interactions and conflicts between concrete design options.

This document is organized as follows: In Chapter 3, we present the generic decision making framework and models based process of the ICEBERG project; in Chapter 4 we discuss in detail the optimal testing resources allocation process.; in Chapter 5 we present the optimal regression functional testing process using coverage and churn metrics; in Chapter 6, we introduce the architectural decision making process. Finally in Chapter 7, we present the conclusions of this work package.

3 A GENERIC MODELS-BASED PROCESS AND DECISION MAKING FRAMEWORK

In the previous work packages of the ICEBERG project, a comprehensive literature review and an industrial survey were carried out to identify the state of the art on:

- Quality management and decision-making needs of software companies,
- Commonly used software tools and commonly collected measures for time, cost and quality
- Potential analysis techniques, methods and tools that could be used for analyzing tradeoffs between cost, time and quality

These were altogether provided a basis when defining a generic models-based process (see Figure 1) and quality decision making framework (see Figure 2) for software companies.

We based the generic process on ISO/IEC 15939 Standard on Software Measurement Process so as to enable companies to be able to use the decision-making framework integrated with their measurement processes. The Models Based Decision Making Process provides a concrete support to software companies when planning their measurement process.

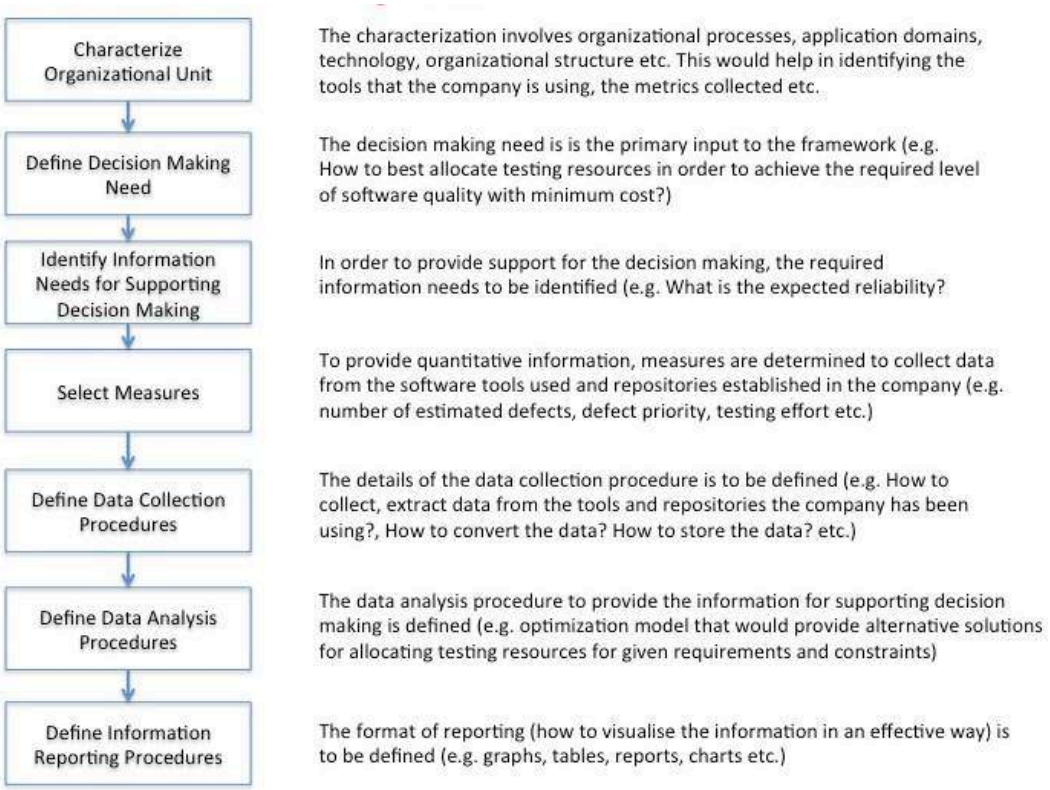


FIGURE 1: A GENERIC MODELS BASED DECISION MAKING PROCESS DEFINITION

By following the steps of this process, the generic decision making framework could be instantiated for a supporting the companies for their specific decision making needs.

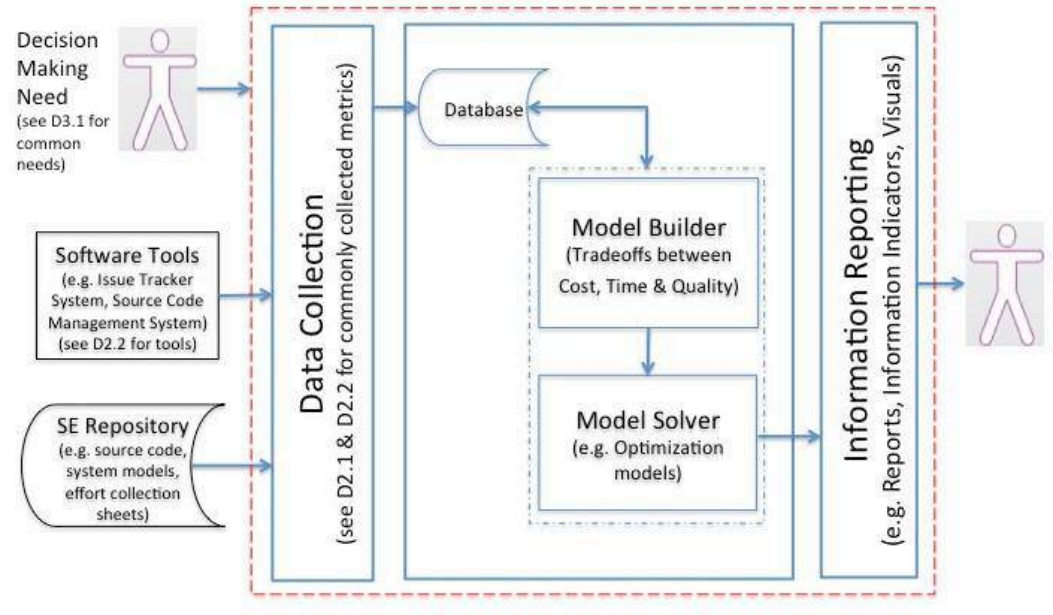


FIGURE 2: A GENERIC DECISION MAKING FRAMEWORK AND ITS ENVIRONMENT

The generic decision making framework comprises a Model Builder, a Model solver and a Database. Primary inputs to this framework include for example, (i) system models (e.g., an UML-based architectural model composed of a Component Diagram, Sequence Diagrams, and a Deployment Diagram), (ii) causes of quality decision-making, and (iii) dependencies among quality decisions, defects issues, cost factor and schedule factor. In particular, we identify: (i) quality decisions (and causes), and (ii) schedule/time/cost-related properties.

The Model Builder generates the analysis model (e.g optimization model) in the format accepted from the solver. The Model solver processes the 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, 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 introduction of 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 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, a set of solutions is devised, 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 would be a chain of quality decisions (leading changes either to the application level or the project management level)..

As shown in Figure Figure 3, a decision-making framework is characterized by input parameters, output parameters, and techniques (e.g., optimization models, algorithms) to make the decisions.

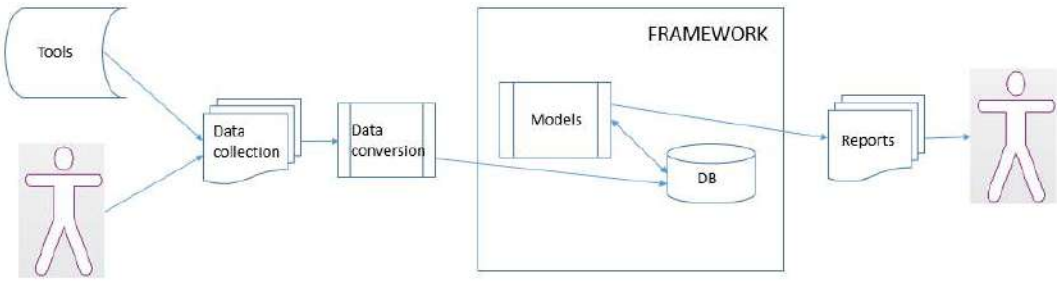


FIGURE 3: A GENERIC DECISION MAKING FRAMEWORK

Below, we provide some examples, which show how to use the models-based process when creating an instant of the decision-making framework for specific decision-making needs. The details of these models are presented in the next chapters: Chapter 4, Chapter 5 and Chapter 6.

3.1 AN EXAMPLE DECISION MAKING FRAMEWORK FOR OPTIMAL ALLOCATION OF TESTING RESOURCES

In this section, we present the framework we developed for making decisions on how to allocate testing resources (see Figure 4). The details of the model are given in Chapter 4.

A primary input to this framework is represented, for example, by from (i) the SRGMs chosen to represent the testing process of the system functionalities, (ii) defect data collection used, for example, to estimate parameters specific to debuggers (e.g., the average amount of bugs that a debugger can fix per man-day), and (iii) requirements on the time and cost of testing (such as on the total amount of testing-effort eventually consumed).

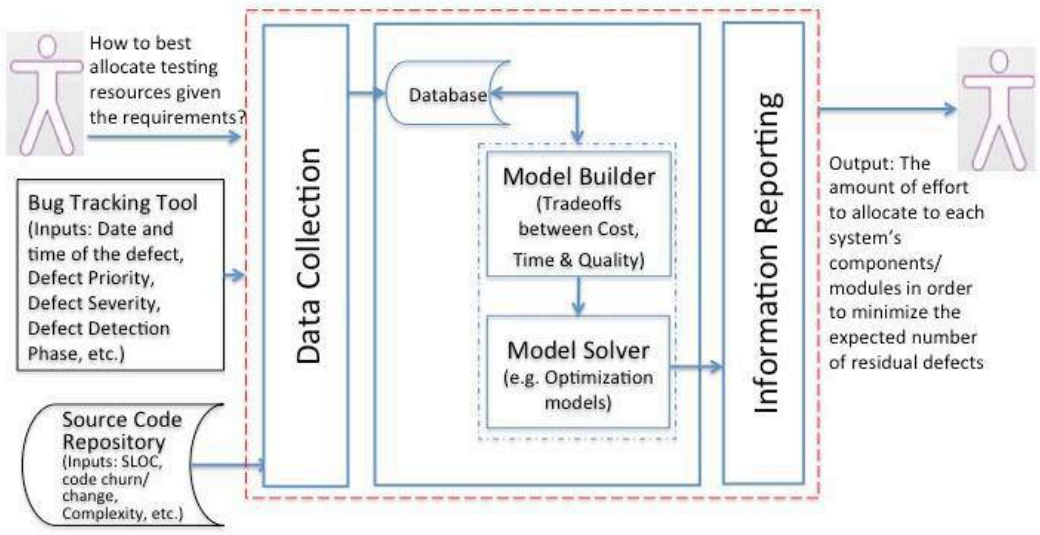


FIGURE 4: AN EXAMPLE FRAMEWORK FOR TESTING RESOURCES ALLOCATION

The Model builder, through a Parameter Specification module, gets input model parameters. After receiving the parameters' specification, the Model builder generates the optimization model in the format accepted by a solver (such as the combination of the NSGA-II algorithm and the MC simulation).

The Model solver processes the optimization model received from the builder and produces the results, which consist, for example, of the testing-effort allocation (i.e., the amount of testing-effort to be performed for the system functionalities) and bug assignment allocation (i.e., the amount of bugs assigned to each of the debuggers).

Inputs

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-months for testing).

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

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

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

TABLE 1: MODEL'S POTENTIAL INPUTS

Source	Measure Category	Measures
Bug Repository	Defect	*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	Product	*Size Measures (LoC, #Req, Function Points), Complexity metrics (McCabe, Halstead's), Source File metrics, code churn/change metrics, version*
Personnel through time sheets or other records	Process	*Testing effort (e.g., man-months dedicated to testing)*
		T Maximum threshold given to the delivery time of the system.

Note that some of the specified analyses are also detailed in the subsequent sections, being this defect analysis model at higher level. Table 2 summarizes the potential outputs of the model.

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).

TABLE 2: MODEL'S POTENTIAL OUTPUTS

Decision Type	Description
Release policy	Quality (reliability) analysis/assessment and time to get a given quality
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
Whether to change the current process based on defect data and if so, how?	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
Whether to improve the debugging process and/or development process	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 3: INPUT-OUTPUT MATRIX DESCRIBING THE POSSIBLE ANALYSES AND OUTPUTS IN RELATION TO PROVIDED INPUTS

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%"
	Defect 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)

	Defect severity/ priority/ reproducibility	Fine-grained Fixing process analysis (analyse potential causes for experienced time to fix)	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 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		Code-defect	How many defect (density) per target;

fix (e.g., source file)		Relationship Analysis	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

In Chapter 4, we discuss how to estimate these parameters by using information collected with a bug-tracking tool (e.g., Jira). We have also instantiated the optimization model for the fault correction with the bug assignment activity prediction, but its elements (e.g., cost function and reliability constraints) combined with the method for uncertainty analysis could be re-used in another phase of the testing process. This adoption may require specializing (appropriately modifying) the model in order to capture typical aspects of the new phase. Testing-effort allocation prediction under testing-effort time/cost and

reliability constraints with uncertain model parameters, for example, could be used for enhancing existing approaches (discussed in Section 4) for scheduling developers/testers to activities to be performed to fix a bug repository.

In Table 4 and Table 5, we discuss in detailed examples for the testing model which we discussed in deliverable D3.1 [1]. In particular, we summarize inputs and outputs of these models.

TABLE 4: MODELS' INPUTS

Model	Input	Reference
Release planning	For each component, Opening time of defects discovered during testing (and/or during peration).	D3.1 – 7.1
Debugging analysis for improved release planning	Input data are the same as the release planning model, as this model is based again on SRGM, augmented by data on closing time of the issues, being the model conceived to include the impact of debugging.	D3.1 – 7.2
Resources allocation	For this model, the required inputs come from the bug-tracking repository 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 managers 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.	D3.1 – 8.1

TABLE 5: MODELS' OUTPUTS

Model	Output	Reference
Release planning	Prediction of the optimal time to release, given a quality to achieve	D3.1 – 7.1
Debugging analysis for improved release planning	Prediction of the optimal time to release, given a quality to achieve and analysis of debugging causes	D3.1 – 7.2
Resources allocation	The amount of effort to allocate to each system's components/modules in order to minimize the expected number of residual defects	D3.1 – 8.1

Table 6 below presents how input information could be represented in a database.

TABLE 6: MODEL'S INPUTS AND THE DATABASE

Model	Input	Database

Release planning	For each component, opening time of defects discovered during testing (and/or during operation).	The table Issue and the relationship Issue-Version allow to obtain information related to opening time of defects discovered during testing. Moreover, relationships in the database allow to get information related to the components, products, projects and companies associated with a certain issue.
Debugging analysis for improved release planning	Input data are the same as the release planning model, as this model is based again on SRGM, augmented by data on closing time of the issues, being the model conceived to include the impact of debugging.	Similarly to the previous decision model, Information related to issues can be found in the database.
Resources allocation	For this model, the required inputs come from the bug tracking repository 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.	Other than the information of the previous two decision modes, information related to the components (modules) can be also found. Such information can be obtained by using the tables Version, Component and Product involved in the relationship Version-Component.

Figure 5 shows the information in and out of the testing decision frameworks listed in Table 4 and Table 5.

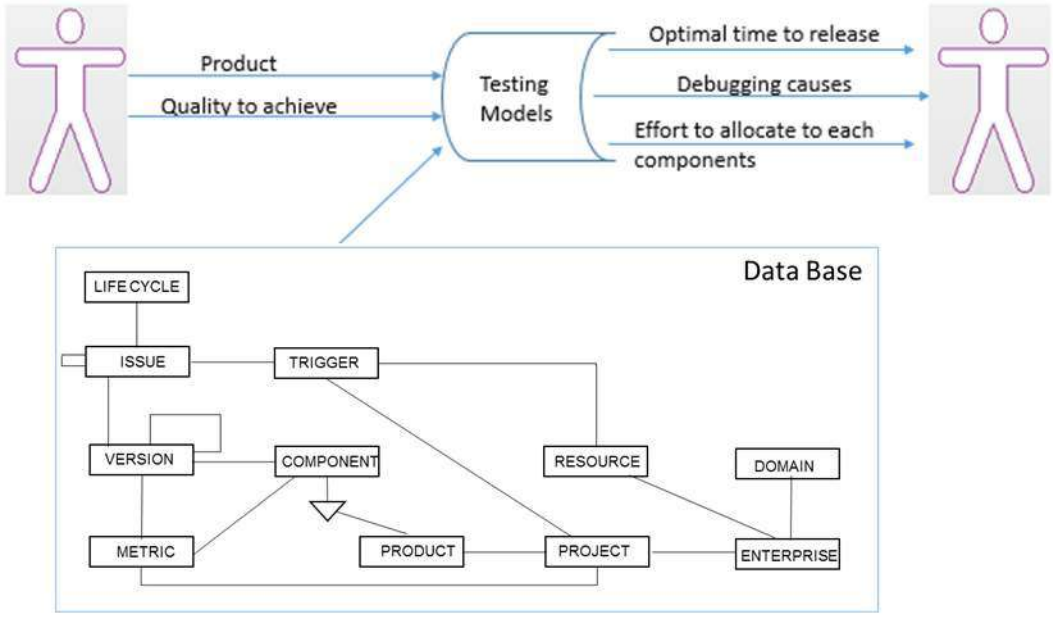


FIGURE 5: INPUTS AND OUTPUTS TO THE DECISION MAKING FRAMEWORK

3.2 AN EXAMPLE ARCHITECTURAL DECISION MAKING FRAMEWORK

Below is an example framework we developed for making decisions on architecture (see Figure 6). The details of the model are given in Chapter 6.

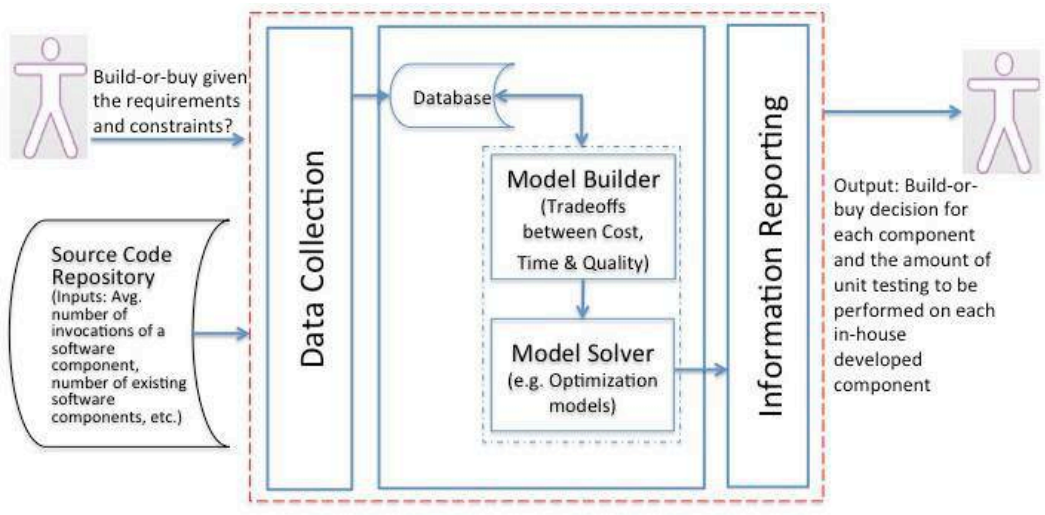


FIGURE 6: AN EXAMPLE DECISION MAKING FRAMEWORK FOR ARCHITECTURAL DECISIONS

In Table 7 and Table 8 we discuss examples of architectural decisions models, which we have discussed in the deliverable D3.1 [1]. In particular, we summarize inputs and outputs of the models.

TABLE 7: MODELS INPUTS

Model	Input	Reference
Build-or-buy decisions models	Average number of invocations of a software component, number of existing software components, maximum number of COTS instances available for each component, number of existing software components, minimum threshold given to the reliability on demand of the system, maximum threshold given to the delivery time of the system, cost of a component instance, delivery time of a component instance, unitary development cost (time) of a component instance, average time required to perform a test case of the instance, testability of a component instance.	D3.1 – 6.2
Quantifying the influence of failure repair/mitigation costs	Average number of invocations of an elementary service across all considered interaction scenarios, minimum threshold given to the reliability on demand of the system, number of nominal services, maximum number of service implementations available for purchase by providers for each nominal service, cost of the service instance, probability of failure on demand of a service instance, unitary development cost of an in-house service, testability of an in-house instance.	D3.1 – 6.2.1
Optimization of adaptation plans with cost and quality tradeoff	Set of new requirements that induce changes in the structural and behavioral architecture of the software system, set of actions that address a certain requirement, average number of invocations of an elementary service, average number of invocations of a new service, number of elementary software services, set of alternative instances for an existing service, cost of a service instance, reliability (availability) on demand of a service instance, response time of a service instance, set of new available services, cost of a new service, reliability (availability) on demand of a new service, response time of a new service, minimum threshold given to the reliability (availability) on demand of the system, maximum threshold given to the system response time.	D3.1 – 6.3

TABLE 8: MODEL'S OUTPUTS

Model	Output	Reference
Build-or-buy decisions models	Build-or-buy decisions for each component and the amount of unit testing to be performed on each in-house developed component	D3.1 – 6.2
Quantifying the influence of failure repair/mitigation costs	Build-or-buy decisions for each service (component as a service) and the amount of unit	D3.1 – 6.2.1

	testing to be performed on each in-house developed service. 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.	
Optimization of adaptation plans with cost and quality tradeoff	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) by removing or introducing interactions between existing services and/or between existing and new services.	D3.1 – 6.3

Table 9 describes how input information of the architectural decision frameworks can be represented in a database.

TABLE 9: MODEL'S INPUTS AND THE DATABASE

Model	Input	Database
Build-or-buy decisions models	Average number of invocations of a software component, number of existing software components, maximum number of COTS instances available for each component, number of existing software components, minimum threshold given to the reliability on demand of the system, maximum threshold given to the delivery time of the system, cost of a component instance, delivery time of a component instance, unitary development cost (time) of a component instance, average time required to perform a test case of the instance, testability of a component instance.	Information related to existing and new components can be found in the database. In particular, for each component instance (represented with tables Version-Component) data are stored. Its information (e.g., related to the delivery time or average time required to perform a test case) are stored in the relationship Metric-Version. Input data inserts by users are related to the number of components, minimum threshold given to the reliability on demand of the system, maximum threshold given to the delivery time of the system.
Quantifying the influence of failure repair/mitigation costs	Average number of invocations of an elementary service across all considered interaction scenarios, minimum threshold given to the reliability on demand of the	Similarly to the previous model, information related to services can be found in the database. In particular, for each service instance (represented with tables

	system, number of nominal services, maximum number of service implementations available for purchase by providers for each nominal service, cost of the service instance, probability of failure on demand of a service instance, unitary development cost of an in-house service, testability of an in-house instance.	Version-Component) data are stored. Its information (e.g., related to the cost of the service instance, probability of failure on demand) are stored in the relationship Metric-Version. Input data inserts by users are related to the number of services, minimum threshold given to the reliability on demand of the system.
Optimization of adaptation plans with cost and quality tradeoff	Set of new requirements that induce changes in the structural and behavioral architecture of the software system, set of actions that address a certain requirement, average number of invocations of an elementary service, average number of invocations of a new service, number of elementary software services, set of alternative instances for an existing service, cost of a service instance, reliability (availability) on demand of a service instance, response time of a service instance, set of new available services, cost of a new service, reliability (availability) on demand of a new service, response time of a new service, minimum threshold given to the reliability (availability) on demand of the system, maximum threshold given to the system response time.	Information related to existing and new services can be found in the database. In particular, for each service instance (represented with tables Version-Service) data are stored. Its information (e.g., related to the reliability, availability) are stored in the relationship Metric-Version. Input data inserts by users are related to the number of services, minimum threshold given to the reliability (availability) on demand of the system, maximum threshold given to the system response time.

Figure 7, Figure 8 and Figure 9 show the inputs and outputs of the architectural decision frameworks listed in the above tables.

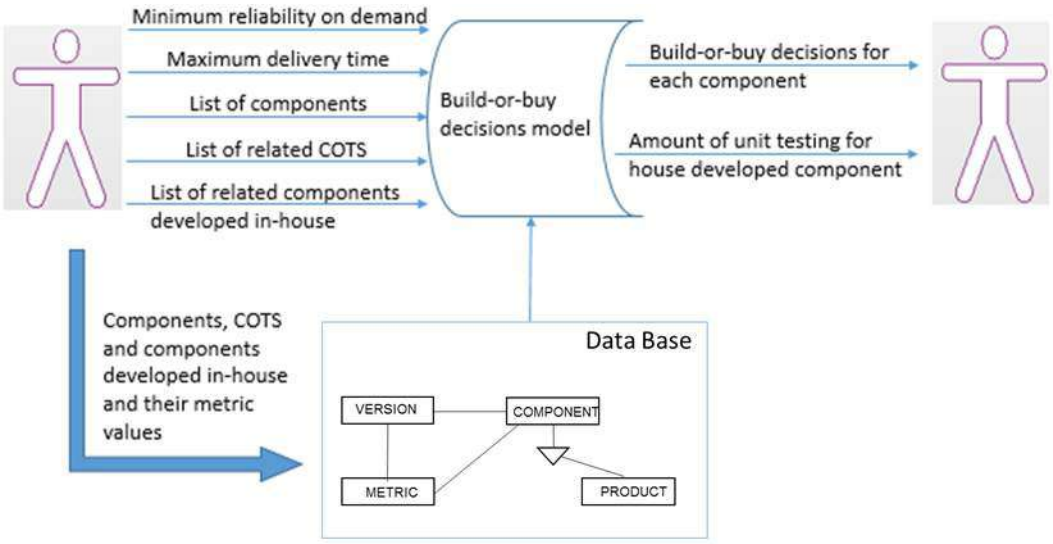


FIGURE 7: INFORMATION IN AND OUT OF THE BUILD-OR-BUY DECISION MODEL

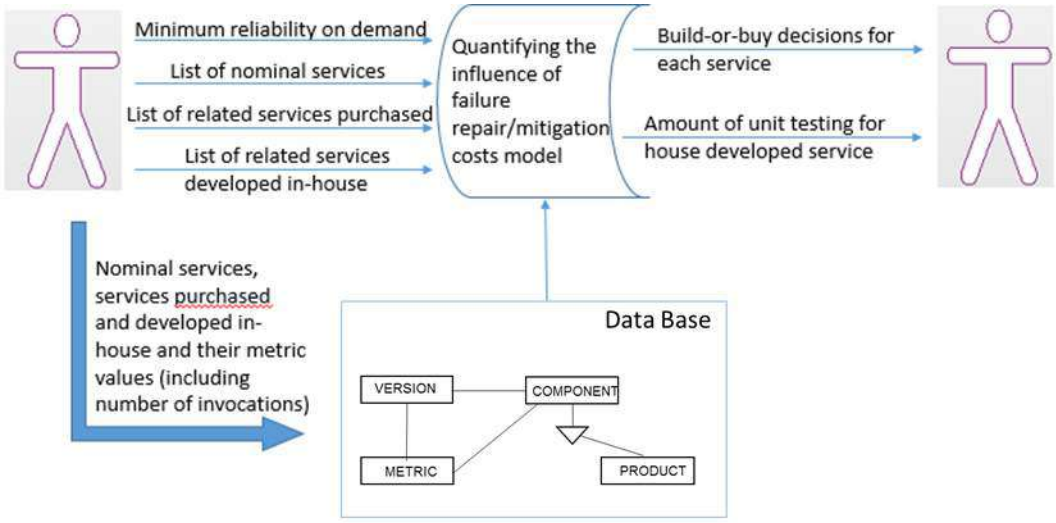


FIGURE 8: INFORMATION IN AND OUT OF THE QUANTIFYING THE INFLUENCE OF FAILURE REPAIR/MITIGATION COSTS MODEL

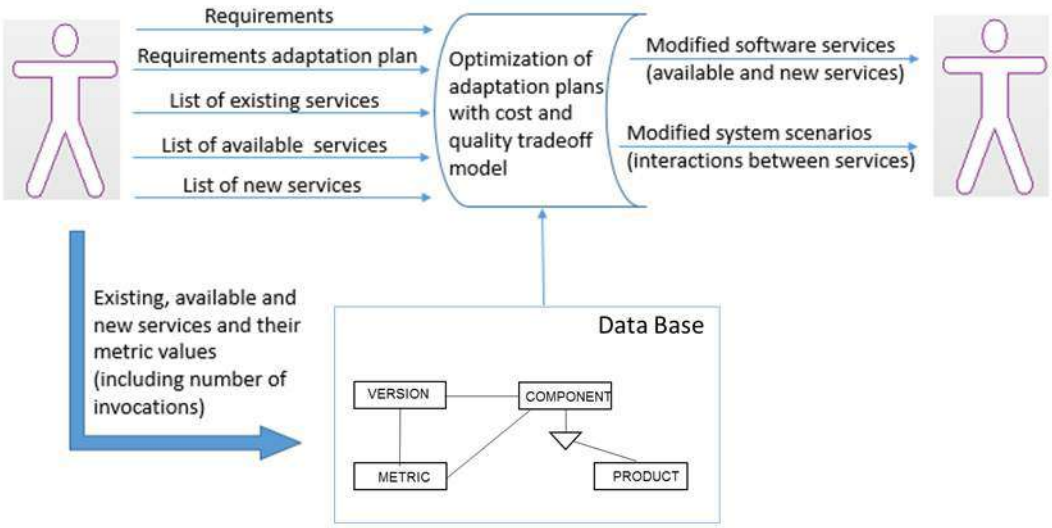


FIGURE 9: INFORMATION IN AND OUT OF THE OPTIMIZATION OF ADAPTATION PLANS WITH COST AND QUALITY TRADEOFF MODEL

3.3 AN EXAMPLE REGRESSION TESTING DECISION FRAMEWORK

In this section, we present the example framework we developed for making decisions on regression testing (see Figure 10). The details of the model are provided in Chapter 5.

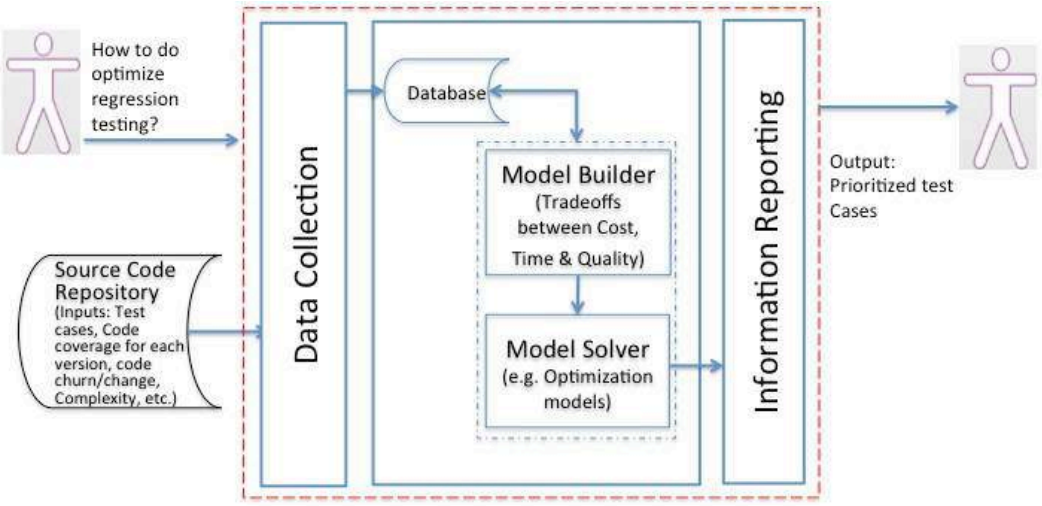


FIGURE 10: AN EXAMPLE DECISION MAKING FRAMEWORK FOR REGRESSION TESTING

In Table 10, Table 11 and Table 12 we discuss examples of regression decision models. In particular, we summarize inputs and outputs of the models.

TABLE 10: MODEL'S INPUTS

Model	Input	Reference
Regression test suite prioritization	Test cases, analysis of code coverage is collected for each of the version of a software product, Churn metrics are collected for each of the version of a software product (e.g., Cyclomatic Complexity, number of added or modified LOC).	More details can be found in Section 5.2

TABLE 11: MODEL'S OUTPUTS

Model	Output	Reference
Regression test suite prioritization	Test cases prioritization.	More details can be found in Section 5.2

TABLE 12: MODEL'S INPUTS AND THE DATABASE

Model	Input	DB
Regression test suite prioritization	Test cases, analysis of code coverage is collected for each of the version of a software product, Churn metrics are collected for each of the version of a software product (e.g., Cyclomatic Complexity, number of added or modified LOC).	Similarly to the architectural decision models, information related to components can be found in the database. In particular, for each component instance (represented with tables Version-Component) data are stored.

Figure 11 shows the information in and out of the regression testing decision framework listed in the above tables.

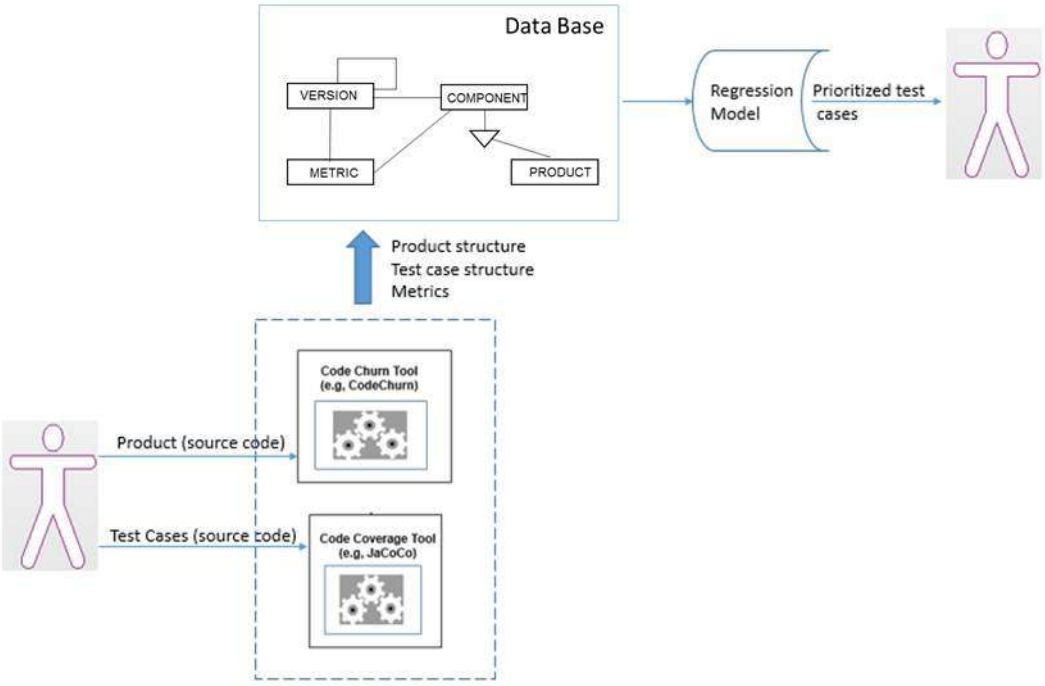


FIGURE 11: INFORMATION IN AND OUT OF THE REGRESSION TESTING MODEL

3.4 DATA GATHERING

In this section, we provide more information for the database, which is to be designed and implemented for collecting the data required by the decision-making models. The ER scheme can be found in deliverable D3.2.

The following diagram illustrates the process of creating the database.

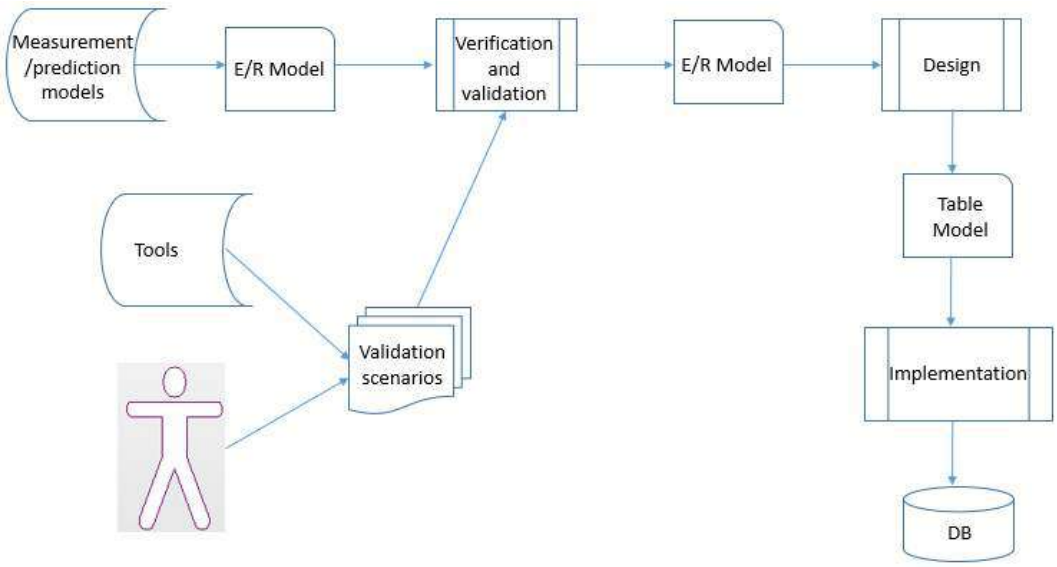


FIGURE 12: THE PROCESS OF DATABASE CREATION

The data collected from the industrial scenarios (provided by our industrial partners) will be used for populating the database, as sketched in Figure 13.

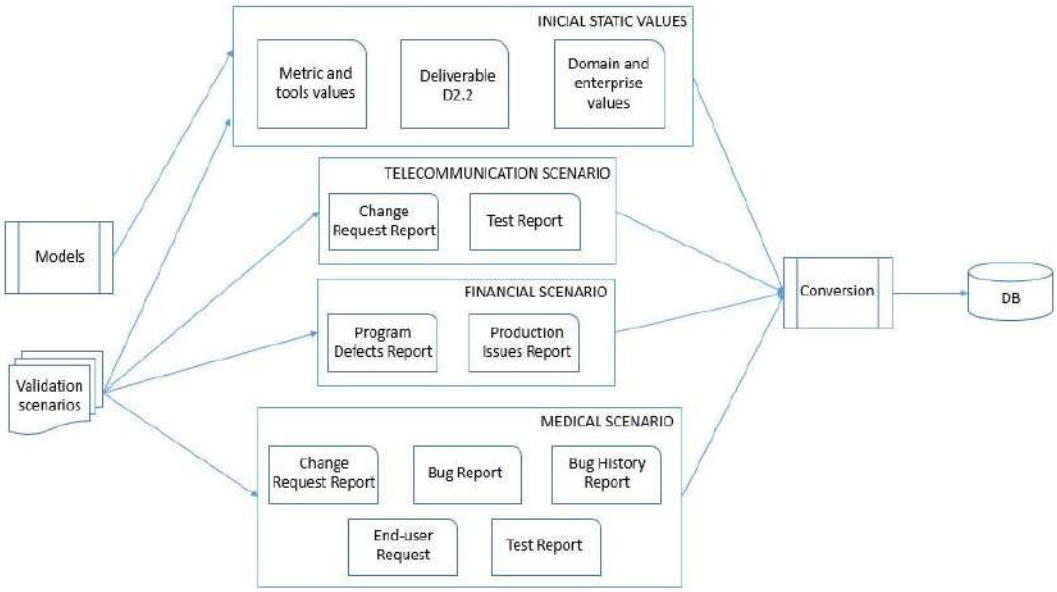


FIGURE 13: THE PROCESS OF DATABASE POPULATION

Information can be categorized in three main categories:

- Metrics
- Products
- Defects

a) Metrics

Figure 14 shows the ER schema related to Metrics information.

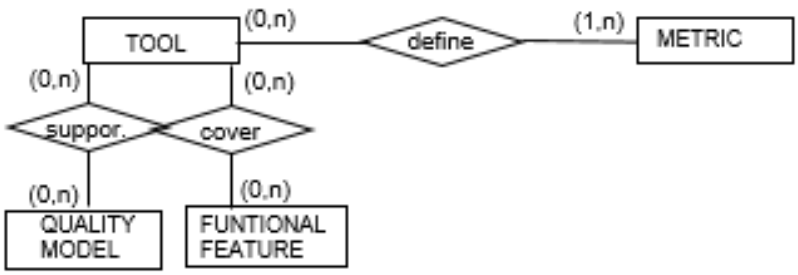


FIGURE 14: ER SCHEMA RELATED TO METRICS

Table *Metric*. This table encompasses the software metrics. In the deliverable D2.2, a quite extensive list of software metrics can be found. Example of metrics is LOC (number of lines of code). Different metrics can be used for different software versions and for different projects. Moreover, two versions of the same components may have different values for the same metric.

The following table summarizes the data related to code churn, which are used for populating the database.

TABLE 13: CODE CHURN METRICS

Classification	Type	Characteristic	Name	Description	Feasibility	UM
Change	Process	Schedule	HOURS	Time in hours to develop/maintain the software system.		number
Change	Process	Frequency	REVISIONS	Number of revisions of a file	good	number
Change	Process	Frequency	REFACTORINGS	Number of times a file has been refactored	good	number
Change	Process	Frequency	BUGFIXES	Number of times a file was involved in bug-fixing	good	number
Change	Process	Size	AUTHORS	Number of distinct authors that checked a file into the repository		number
Change	Process	Size	LOC_ADDED	Sum over all revisions of the lines of code added to a file		number
Change	Process	Size	MAX_LOC_ADDED	Maximum number of lines of code added for all revisions		number

Change	Process	Size	AVE_ LOC_A DDED	Average lines of code added per revision		number
Change	Process	Size	LOC_DE LETED	Sum over all revisions of the lines of code deleted from a file		number
Change	Process	Size	MAX_L OC_DEL ETED	Maximum number of lines of code deleted for all revisions		number
Change	Process	Size	AVE_LO C_DELE TED	Average lines of code deleted per revision		number
Change	Process	Size	CODEC HURN	Sum of (added lines of code – deleted lines of code) over all revisions		number
Change	Process	Size	MAX_C ODECH URN	Maximum CODECHURN for all revisions		number
Change	Process	Size	AVE_C ODECH URN	Average CODECHURN per revision		number
Change	Process	Size	MAX_C HANGE SET	Maximum number of files committed together to the repository		number
Change	Process	Size	AVE_C HANGE SET	Average number of files committed together to the repository		number
Change	Process	Size	AGE	Age of a file in weeks (counting backwards from a specific release)		number

Change	Process	Size	WEIGHTED_AGE	$\frac{\sum_{i=1}^{N} Age(i) \times LOC\_AD}{\sum_{i=1}^{N} LOC\_ADDE}$ (Pg.42 Doc. D2.2)		number
Change	Resource	Effort	PERSON-HOUR	Cost per hour to develop/maintain the software system.	high	euro
Change	Resource	Effort	PERSON-DAYS	Cost per day to develop/maintain the software system.	high	euro
Change	Resource	Cost	MONEY	Money value (per hour/day/week/month) average or differentiated by employee.	high	euro
Source	Product	Size	MB	Megabyte	high	number
Source	Product	Size	FP	Function Point	high	number
Source	Product	Structure	WMC	Weighted Method Count	high	number
Source	Product	Structure	DIT	Depth of Inheritance Tree	high	number
Source	Product	Structure	RFC	Response For Class	high	number
Source	Product	Structure	NOC	Number Of Children	high	number
Source	Product	Structure	CBO	Coupling Between Objects	high	number
Source	Product	Structure	LCOM	Lack of Cohesion in Methods	high	number
Source	Product	Structure	FAN_IN	Number of other classes that reference the class	high	number

Source	Product	Structure	FAN_OUT	Number of other classes referenced by the class	high	number
Source	Product	Structure	NOA	Number of attributes	high	number
Source	Product	Structure	NOPA	Number of public attributes	high	number
Source	Product	Structure	NOPRA	Number of private attributes	high	number
Source	Product	Structure	NOAI	Number of attributes inherited	high	number
Source	Product	Size	LOC	Number of lines of code	high	number
Source	Product	Structure	NOM	Number of methods	high	number
Source	Product	Structure	NOPM	Number of public methods	high	number
Source	Product	Structure	NOPRM	Number of private methods	high	number
Source	Product	Structure	NOMI	Number of methods inherited	high	number
Source	Product	Structure	AHF	Attribute Hiding Factor	high	percentage
Source	Product	Structure	MIF	Method Inheritance Factor	high	percentage
Source	Product	Structure	AIF	Attribute Inheritance Factor	high	percentage
Source	Product	Structure	MHF	Method Hiding Factor	high	percentage
Source	Product	Structure	POF	Polymorphism Factor	high	percentage
Source	Product	Structure	COF	Coupling Factor	high	percentage

Source	Product	Structure	SIX	Specialisation Index per Class	high	percentage
Source	Product	Structure	CCN	Cyclomatic complexity	high	number
Source	Product	Structure	LOCM4	Lack Of Cohesion of Methods version 4	high	number
Source	Product	Structure	Package tangle index	cyclical dependencies between packages and files		percentage
Source	Product	Size	PLOC	Number of physical lines of code	high	number
Source	Product	Size	LLOC	Number of logical lines of code	high	number
Source	Product	Structure	NOC	Number of class	high	number
Source	Product	Structure	NOP	Number of packages	high	number
Source	Product	Structure	NOF	Number of files	high	number
Source	Product	Structure	BRANCHES	Number of branches (for all if and switch statements)	high	number

Table *Tool*. This table encompasses the tools for metrics evaluation (e.g., the Sonar tool). A raw list of these tools can be found in the deliverable D2.2. A tool can be used to collect several metrics. Therefore, there is a relationship N:N between the tables *Metric* and *Tool*.

Examples of tools are shown below in Table 14.

TABLE 14: EXAMPLE TOOLS

Name	Description	YearFirst Version	YearLast Version
Jdepend	JDepend traverses Java class file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively.	2006	2014
JCSC	JCSC is a powerful tool to check source code against a highly definable coding standard and potential bad code. It is a highly configurable checking tool for your Java source code. It checks the compliance to a defineable coding standard like naming conventions and code structure. Also signs of bad coding, potential bugs are found.	2002	2005
QALab	QALab consolidates data from Checkstyle, PMD, FindBugs and Simian and displays it in one consolidated view. QALab keeps a track of the changes over time, thereby allowing you to see trends over time. You can tell weather the number of violations has increased or decreased - on a per file basis, or for the entire project. It also plots charts of this data.	2006	2006
CKJM	CKJM calculates Chidamber and Kemerer object-oriented metrics by processing the bytecode of compiled Java files. The program calculates for each class the following metrics: weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, response for a class, lack of cohesion in methods, afferent couplings and number of public methods	2005	2012
Panopticode	The Panopticode project provides a standardized format for describing the structure of software projects and integrates metrics from several tools into that	2007	2007

	format. Reporting options provide correlation, historic analysis, and visualization. Panopticode uses Tree Maps to display the code complexity and coverage.		
Same	Same is a tool to find duplicate lines in multiple text files. Very useful to find and fix copy-and-paste programming. It has been designed to be simple, portable, and fast.		2001
FindBugs	It uses static analysis to look for bugs in Java code. Potential errors are classified in four ranks: scariest, scary, troubling and of concern. This is a hint to the developer about their possible impact or severity.	2007	2015
JavaNCSS	JavaNCSS is a simple command line utility which measures two standard source code metrics for the Java programming language. The metrics are collected globally, for each class and/or for each function. It can optionally present its output with a little graphical user interface.	1997	2009
PMD/CPD	PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports Java, JavaScript, PLSQL, Apache Velocity, XML, XSL. CPD is a copy-paste-detector. CPD finds duplicated code in Java, C, C++, C#, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Ruby, Scala, Objective C, Matlab, Python, Go.	2002	2015
Xradar	XRadar is an open extensible code report tool currently supporting all Java based systems. The batch-processing framework produces HTML/SVG reports of the systems current state and the development over time - all presented in tables and graphs. It gets results from several open source projects and a couple of in house grown projects and presents the results as massive unified html/svg reports.	2008	2009

Checkstyle	Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code. It is highly configurable and can be made to support almost any coding standard. It can find class design problems, method design problems. It also has the ability to check code layout and formatting issues.	2007	2015
Sonar	It is an open source platform for continuous inspection of code quality. Offers reports on duplicated code, coding standards, unit tests, code coverage, complex code, potential bugs, comments and design and architecture. Records metrics history and provides evolution graphs and differential views.	2007	2015
Classycle	Classycle's Analyser analyses the static class and package dependencies in Java applications or libraries. It is especially helpful for finding cyclic dependencies between classes or packages. Classycle's Dependency Checker searchs for unwanted class dependencies described in a dependency definition file. Dependency checking helps to monitor whether certain architectural constrains are fulfilled or not.	2003	2014
Jlint	Jlint will check your Java code and find bugs, inconsistencies and synchronization problems by doing data flow analysis and building the lock graph. Jlint is extremely fast - even on large projects, it requires only one second to check all classes. It is easy to learn and requires no changes to the class files.	2004	2011
Sonar Plugins	Sonar includes several plugins such as language plugins, plugins for developer tools, governance, integration, autentication and authorization, additional metrics, SCM engines, external analizers, visualization, reporting, etc.	2014	2015

Squale	Assists developers in improving the code of their projects. Helps project managers to meet quality requirements for their applications. Gives top-managers dashboards to monitor the overall health of their information system. Works on enhanced quality models. Helps assessing software quality and improving it over time.	2009	2011
JaCoCo	JaCoCo is an open source toolkit for measuring and reporting Java code coverage. It offers line and branch coverage. JaCoCo instruments the bytecode while running the code. To do this it runs as a Java agent, and can be configured to store the collected data in a file, or send it via TCP.	2009	2015

Relationship *Tool-Metric*. This relationship is used for obtain metrics values. In particular, this relationship can be obtained from Table 19 of the deliverable D2.2. For example, the JaCoCo tool (see Section 6) provides code coverage metrics.

Table *Qualitymodel.* This table encompasses quality models, for example ISO 9126 (see Deliverable 2.2). A tool may be related to one (or more) quality models. Therefore, there is a relationship N:N between the tables *Qualitymodel* and *Tool.*

Examples of quality models are:

Name	Description
ISO 9126	International standard for the evaluation of software quality. Its fundamental objective is to address some of the well known human biases that can adversely affect the delivery and perception of a software development project.The standard is divided into four parts: quality model, external metrics, internal metrics and quality in use metrics. It has been replaced by ISO/IEC 25010:2011
ISO 25010	This quality model determines which quality characteristics will be taken into account when evaluating the properties of a software product. The considered characteristics are: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability.
SQUALE	It is inspired by the ISO 9126 standard and introduces a new level for the assessment of practices in the hierarchy of factors, criteria, and measures. It allows one to determine the quality of a project and control its evolution during the maintenance of a project, preventing deterioration. The Squale model stresses bad quality instead of averaging the quality in order to quickly focus on the wrong parts. It uses a set of measures

	combined into practices using formulae which take into account company standards and project technical specificity. Practice weights are customized with respect to these overall constraints.
SIG	It is based on best practices and defined standards, such as ISO/IEC 25010. The SIG model offers an efficient, simple and structured way to gain objective insight in the quality of performance by evaluating both the process and the product. The result is a score from one to five stars, where more stars correspond to a higher quality. One of the key aspects of the model is 'observability', a property that discusses to what extent performance characteristics in a system can be measured and assessed.

Relación *Tool-QualityModel*. This relationship is derived by applying the quality models.

Entity *Functionalfeature.* This table encompasses the tasks that metric tools perform (e.g., the *Data acquisition* task). In the deliverable D2.2, a description of the main tools' tasks is provided. A tool can perform one or more tasks. A specific task can be performed by more than one tool. Therefore, there is a relationship N:N between the tables *Tool* and *Functionalfeature.*

First, it is populated the following table:

Name	**Description**
Data acquisition	Set of methods and techniques for obtaining necessary data for measurement
Analysis of measures	Ability to store, retrieve, manipulate and perform data analysis
Data presentation	Formats to generate the obtained documentation

Relationship *Tool-FunctionalFeature*. This relationship can be obtained from Table 20 of deliverable D2.2.

b) **Products**

Figure 15 shows the ER schema related to Products information.

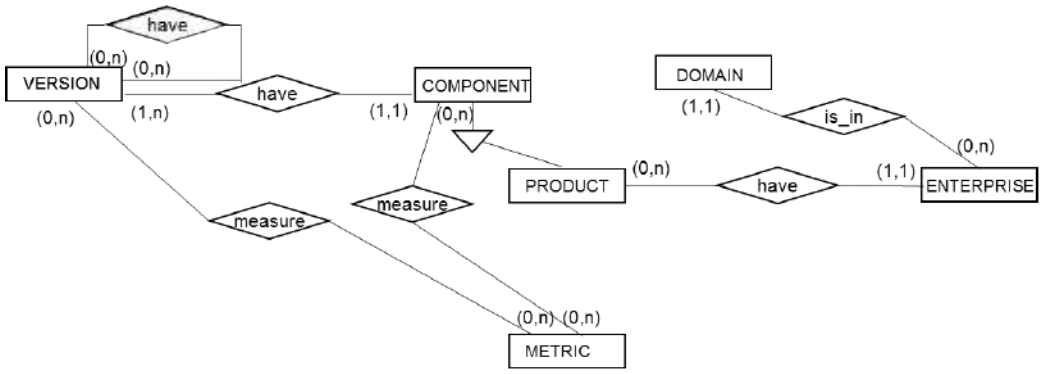


FIGURE 15: ER SCHEMA RELATED TO PRODUCTS

Table *Domain*. This table collects information related to the application domain of the company (e.g, medical, telecommunications, financial, which are the ICEBERG project's scenarios application domain).

Table *Enterprise*. An occurrence of this table represents an organization responsible for the software development or maintenance. Data privacy is considered by using appropriates measures (e.g., by inserting names of scenarios as ScenarioM, ScenarioT, ScenarioF).

Table *Product*. A software product is a component that results of a composition of one or more components. There is a hierarchy/aggregation relationship between a product and a component. For each product, information related to its providers are stored.

Table *Component* and Table *Version*. A component is a self-contained deployable software module containing data and operations, which provides/requires services to/from other components. Different versions may be available for one component. A component version is a specific implementation of a component. A component version can be involved in different software product versions and in others component versions. Defects are related to products, components or component's versions. For each scenario, information of its components (and the related versions) are stored (e.g., number of bugs, issues, etc).

Relationship *Version-Version*. This relationship is used to determine the structure of a product. In particular, the decomposition of components/versions in sub-components (versions) is modelled.

Relationship *Metric-Version*. This relationship is used to determine metric values of the component versions.

Relationship *Metric-Component*. This relationship is used to determine metric values of the components.

c) Defects

Figure 16 shows the ER schema related to Defects information.

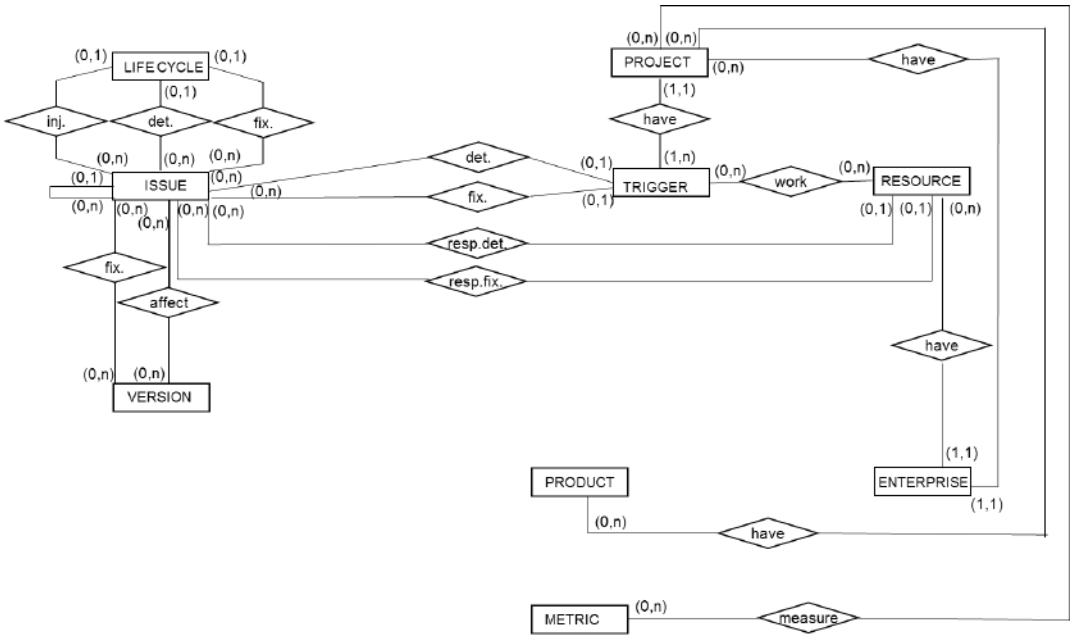


FIGURE 16: ER SCHEMA RELATED TO DEFECTS

Table *Project*. Data related to specific activities, which a software company conduced for developing a project, are stored. Such information is collected with respect to the scenarios. For example, for the ScenarioT, it is stored information related to two products developed in two different projects. Generic projects (for developing or maintain a software systems) are created, which can be used to insert data of new further scenarios.

Relationship *Product-Project*. This relationship allows obtaining information related to the products and the projects (i.e., project specific activities).

Relationship *Metric-Project.* This relationship allows to obtain metrics related to a single project. More specifically, if a project is related to one product, then metrics values of the product will be stored. The current version of the project is also stored.

Table *Resource.* This table encompasses the people involved in the different activities. Several people may be involved in the same *trigger*. For each activity, the working hours of the people can be stored (assuring later analysis of cost/effort data). In particular, for each scenario, information related to the people involved are stored. Data will be stored by appropriating adopting privacy mechanisms (e.g., for people name will be used a nickname).

Table *Lifecycle*. This table is related to the software life cycle. Different phases are typically involved in a software life cycle, such as the requirements, design, and testing phases.

Table *Issue*. This table encompasses defects, which have been detected or fixed (see deliverable D3.1 for more details). One defect can be associated to one *version* of a software component. For each scenario, information related to the issues found during testing activities (or at operational time) will be stored. For example, opening time (and closing time) of the issue will be collected. Other data are related to the severity, priority, type of bug, or the current state of the issue (e.g., opened, closed, and assigned).

Relationship *Issue-Version*. A defect impact (affected and/or fixed) to one or more versions of a software component. This information is stored with respect to each of the scenarios.

Relationship *Issue-Issue*. A defect may occur again even after the defect is fixed. The reopen defect issue has to be related to the original defect.

Table *Trigger*. This table encompasses the work tasks to be performed to address an *issue*'s occurrence, such as the execution of a test case. A defect (related to the *Issue* entity) may be detected during the execution of a trigger (e.g., during the testing activity) or may be fixed by a trigger. If a scenario does not provide details about triggers, then a generic trigger is created in order to store data. For each project, attributes for its triggers will be stored (e.g., NumTotal, NumPassed, and NumFailed).

Relationship *Trigger-Resource*. Information related to people involved in the single projects (and their specific triggers) will be stored.

4 OPTIMAL ALLOCATION OF TESTING RESOURCES

The allocation of testing resources of large software systems is a complex task, mostly because it requires models that encompass the composition of test process properties into system properties. As software is used more and more in business-critical and safety-critical applications, it is important to prevent the realization of software with poor software quality. The reliability of a large-scale software system is given by the composition of system functionalities (modules, sub-systems, etc) reliabilities; therefore, the system reliability is a function of the detection ability of the testing process of each of the system functionalities (modules, sub-systems, etc).

Typically project managers' decisions span from the identification of the most important system functionalities (e.g., the ones with the biggest safety impact, or the largest financial impact on users) through resource scheduling to staff assignment [2]. In fact, the majority of software projects today are embedded in dynamic contexts, where requirements, environment assumptions, and usage profiles continuously change. Therefore, in the last few years, development processes have primarily focused on the maintenance phase, due to the frequent changes required by software after the deployment phase. In this work, we focus on resource allocation, which is highly relevant in testing process, and is typically a time-consuming and tedious task. It is well worth optimizing the allocation scheme [3]: although testing resources can be allocated in rather simple ways (e.g. average allocation, random allocation, and proportional allocation), an optimal allocation scheme may lead to significant improvement in terms of the reliability of a software system [4].

Any combination of testing allocation decisions may have a considerable impact on the cost, time and software quality. For these multi-attribute problems, there is usually no single global solution, and the generation and evaluation of alternatives can be error prone and lead to suboptimal decisions, especially if carried out manually by test/project managers. Therefore, tools that support decisions strictly related to meet quality/time requirements, while keeping the costs within a predicted budget, would be very helpful to the project managers' tasks.

The presence in the market of standard off-the-shelf components/services has drastically changed in the last decade the development process of large-scale systems. Mission-critical large-scale systems, for example, are developed in a highly modular way, adopting a strong component-based approach to foster reuse and a build-by integration approach [5]. Although several approaches have been introduced in the last few years to address these issues, the tradeoff analysis among quality, cost, and time has not yet been studied enough. In fact, 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 functionality (software module) more "critical" than another, therefore deserving more testing. As there may be relevant differences among functionalities (modules) in terms of quality e.g., because they come from

different teams (internal or external in case of outsourcing), or they are based on different programming paradigms - their defectiveness can vary significantly [6].

The tradeoff analysis results may be strongly affected by parameter uncertainties; in fact, the software testing activity is fraught with a not negligible uncertainty relates to values of parameters such as operational profile, the expected number of initial faults, the fault detection rate per unit testing-effort (SRGMs input), fault fixing time, etc. The propagation of this uncertainty on the objective function and the constraints should be analyzed. Typically, existing works perform the sensitivity analysis of optimal resource allocation problems [7], [8] with respect to those parameters deemed critical, such as the expected initial faults, the fault detection rate and cost of correcting an error in testing and operational phase on the optimum allocation. Because parameters are estimated based on the available data (e.g., parameters of a SRGM are estimated based on the available failure data, which is often sparse [7]), their estimation only represent approximations of parameters. As a consequence, parameter estimation plays a critical role in accurately describing testing resource allocation process through optimization models.

The goal of our work is to assist test/project managers in the decisions on how to effectively distribute the resources available for testing. Such assistance aims to take into account several quality attributes of the testing process, i.e., cost (such as that one to correct a bug during testing, or the cost of testing per unit testing-effort expenditures), time (e.g., the time to detect and fix a bug/defect/fault1), and reliability. In particular, we explicitly consider uncertainty in the testing process in order to evaluate the robustness of the testing resource allocation. Robustness refers to the ability to tolerate uncertainty in the intrinsic input parameters of the testing process. We deal with input parameter uncertainties, and model each uncertain parameter as a random variable whose variability is characterized by its continuous or discrete distribution. We present a Monte Carlo (MC) simulation-based approach to systematically assess the robustness of a resource allocation alternative despite its uncertainty. MC is a well-assessed method for uncertainty analysis. Examples of its adoption can be found in different areas of the scientific literature. Its effectiveness and efficiency have, for example, already been demonstrated in the works [9] [10] for handling parameter uncertainties in the performance (and reliability) modeling and analysis process of software architectures.

More specifically, we provide an automatic optimization process for dynamically allocate testing resources to software modules (functionalities) based on trade-offs among software quality, cost, and schedule/time requirements. Dynamic refers to the ability of using testing data (i.e., bug reports[2]) 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. Our approach consists in formalizing the decision problem in terms of system quality and testing cost/time requirements, to elicit and represent uncertainties as probability distributions, to simulate the impact of resource allocation alternatives

[1] The term fault (defect/bug) is preferred in the fault tolerance (software engineering) community; here, we use them as synonymous.

[2] A bug report is also called a ticket, an issue, an incident, a fault (defect) report, a maintenance request, etc.

on system quality and testing cost/time through MC simulations, and to shortlist a set of alternatives using Pareto-based multi-objective optimization techniques.

Our optimization method combines the application of both metaheuristic search techniques and MC simulations. In particular, we have chosen to adopt evolutionary algorithms because they have been reported to perform better than some other techniques used for solving the testing resource allocation problem (as remarked in [3]). These types of metaheuristic algorithms possess the strong capability of global search, and are usually not very sensitive to initial solutions. On the contrary, these characteristics represent drawbacks that are common among the alternative approaches adopted for solving testing allocation problem. Evolutionary techniques' effectiveness has also been demonstrated on a large spectrum of problems in the reliability optimization field, such as resource management and task partition in grid systems, redundancy allocation, and reliability optimization of weighted voting systems [3].

In a limited testing budget (and time), an important challenge to address is a tradeoff between (i) allocating resources to functionalities (software modules) where testing will have the highest detection power, and (ii) maximizing the number of bugs that can be fixed in available time. This challenge stems from our experience in testing industrial health care systems, in collaboration with our partner. This problem is currently relevant for our industrial partner in particular, and the health care domain in general due to its high variability in requirements and design. In fact, medical procedures and uncertainty in patient behaviors require stochastic analysis, and complex decisions under uncertainty are notably made about the cost-effectiveness of new medical treatments based on the results of clinical trials [11].

In summary, our main contributions are:

- An approach implemented as an optimization framework for dynamically modeling: (i) fault detection and correction processes of systems functionalities (modules) through the SRGMs that best fit the actual testing data, (ii) testing cost/time constraints, and (iii) parameter-specific uncertainties phenomenons. So that the systems functionalities (modules) with shorter time (budget) are tested and that reveled bugs are fixed earlier.
- The maximization of the testing process's effectiveness by predicting the fault correction process as a function of the bug assignment process. More specifically, we predict the ability of the debuggers/testers to correct faults. We use bug reports (collection of fixed and not-fixed bugs) in order to predict debugging performance. In fact, the scheduling of debuggers to bug-fixing activities should not be performed only during system testing, when a new bug is reported and has to be assigned to a developer/debugger for fixing it (see the typical steps of a bug-tracking system such as Bugzilla [12]). If the bug assignment would be limited to the testing activity's execution, then it would be difficult to find bug-fix solutions that are relevant to a given testing situation (e.g., that exactly match the budget and time requirements). We claim that the bug assignment (typically a time-consuming and tiresome process in large software projects [13]) may be a key factor for a trustworthy prediction of the fault correction process of the single functionalities (software modules), as well as of the reliability of the whole system.

In Sections 4.1 and 4.2, we present an overview of the dynamic testing resource allocation framework; in Section 4.3 we provide the formulation of the optimization model that represents the core of our approach; in Section 4.4 the achieved results on the Medical Company scenario (see Deliverable D2.2 [95]) are presented; Section 4.5 introduces related work and discusses the novelty of our contribution.

4.1 OPTIMAL TESTING RESOURCE ALLOCATION PROCESS

We defined a process, which helps in dynamically allocating testing resources to software functionalities. Dynamic refers to the ability 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.

The defined process is based on a multi-objective optimization model combined with a Montecarlo simulation strategy, aiming to maximize the quality of a given software (i.e., in terms of number of detected and corrected faults), based on the trade-offs among system reliability, testing time, and testing/debugging cost.

We hereafter denote the three objectives to be pursued as: FCO (Fault correction process' Effectiveness Objective), to maximize; TTO (Testing Time Objective), to minimize; TCO (Testing-effort Cost Objective), to minimize. The output of our process is a solution (i.e., individual in the NSGA-II terminology) providing (i) the testing effort to be spent for each system functionality, (ii) the number of debuggers being assigned to each functionality, (iii) the hours of each debuggers to the functionalities. A solution is also characterized by the fitness, i.e., the triple composed by the values of FCO, TTO, and TCO that are obtained by the solution.

In the following, we provide a high-level overview of the proposed process.

- **SRGM Construction.** The first phase of the process is obtaining the module-level SRGMs[3] that characterize the testing progress of each functionality. Differently from previous work on SRGMs-based allocation (e.g., [4], [30], [107], [108], [30]), we do not assume any prior specific SRGM, but we infer the most suitable for each functionality.

  More important, the process includes the possibility to dynamically select the best SRGMs during testing as fault detection data become available, whenever historical data are unavailable or unreliable. The steps of the SRGM construction are shown in Figure 17.

[3] For this work, a module is a functionality: in the following , the two terms are used as synonymous if not differently specified.

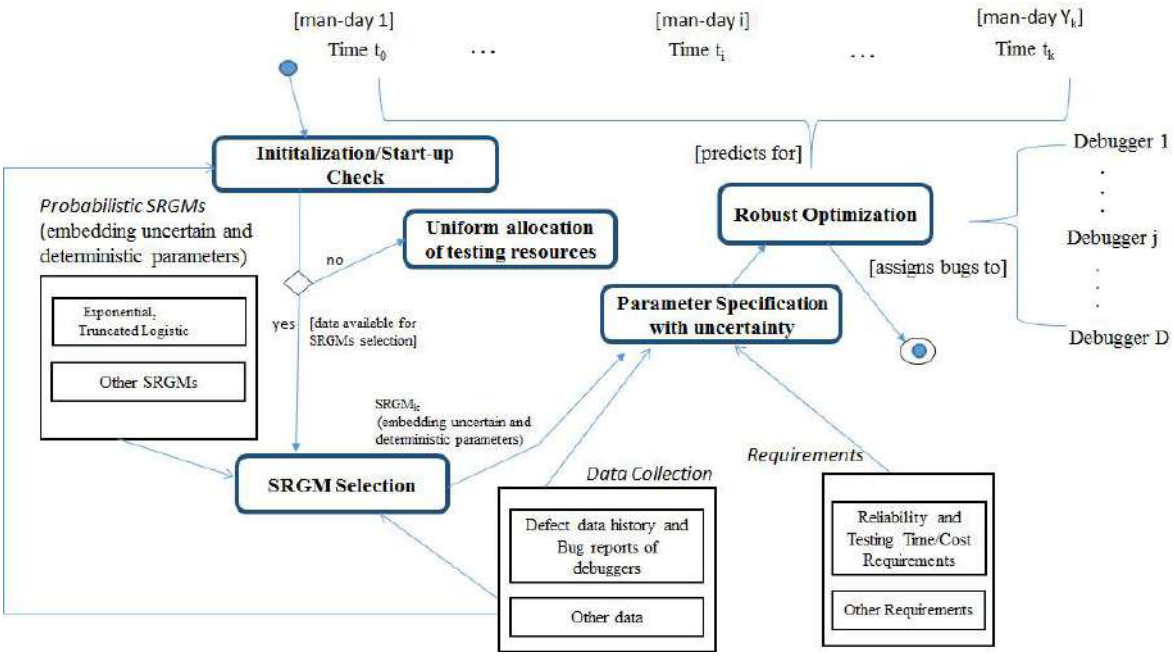


FIGURE 17: HIGH-LEVEL TESTING RESOURCE ALLOCATION PROCESS OVERVIEW

- **Data Gathering.** Let F denote the set of functionalities to test. At the beginning of the process application, i.e., t0, there are two possible cases for a given functionality: i) historical data about testing conducted on that functionality are available (or testing data of another system including the same functionality, or also testing of a previous version of that functionality) ii) no previous data are available.

  In the former case the data (in particular, the fault detection times) can be used to fit an SRGM for the functionality among a list of SRGMs. In the latter case, i.e., without any additional information to prioritize the testing efforts at t_0, the initial resource allocation is done uniformly to all functionalities: once the testing starts, the new data can be progressively used to fit the SRGMs.

  It should be observed that the former case allows running the optimal allocation before the beginning of the testing activities; however, it requires historical data. The latter case uses the data generated during the ongoing testing process (hence, more accurate), but the optimal allocation algorithm can be run only when enough data are available to build the SRGMs. Running the optimal allocation dynamically during testing (possibly, several times) yields to more accurate results, but might be less useful if run too late (since the suggested allocation would apply just to the remaining testing time) [5].

- **Validity check.** To assign a SRGM to a functionality, a validity check is performed to evaluate if data (either historical data or collected during testing) can be fitted in a satisfactory way. Each functionality is fitted by means of every available SRGM among a set of SRGMs the tester wish to try. Fitting is performed by means of the EM algorithm [14], which

provides the best fitting parameters for a given dataset and SRGM. On each SRGM, it is run a goodness of fit (GoF) test, by means of the one-sample Kolmogorov-Smirnov (KS) test (with 95% confidence level) for comparison of samples with a reference probability distribution. If the test is satisfied for at least one SRGM, it means that the testing dataset can be said, with 95% of confidence, to come from that SRGMs distribution.

Once the validity check is passed, we have, in general, a set of SRGMs that satisfy the KS test for one given functionality; these are said to be statistically valid SRGMs. Among them, the best one will be selected according to the next step3.

- **SRGM selection.** The input to this step is the set of statistically valid SRGMs for each functionality. They are compared in terms of fitting ability and the best one is selected. We adopt a common goodness-of-fit measure based, the Akaike Information Criterion (AIC). The SRGM model with the lowest AIC value is preferred, denoting the minimal information loss that we incur by selecting that model. This way, each functionality is assigned with the best fitting SRGM based on real testing data.

❖ **Parameters Specification.** The second phase of our process deals with the specification of parameters, and the management of the uncertainty. Parameters are split into deterministic and uncertain. Deterministic parameters (e.g., desired threshold of reliability, available testing budget, cost of a tester and a debugger per hour) do not need any preliminary treatment. Uncertain parameters (e.g., SRGM parameters, average fixing time, usage profile) are treated by means of a Montecarlo-based strategy aimed at providing the robustness of the solution against the variability of the parameters.

Examples of uncertain parameters (other parameters are listed in the following Section) are the SRGM ones. Their values are, in fact, derived by fitting a dataset, and represent just one of the potential set of values tied to the specific "instance" of data observed from testing – namely, repeating testing on that functionality would give different results, as testing is a random process.

Uncertainty is addressed by considering the value of a parameter as a sample of a probability distribution, similarly to works on architectural solution optimization [9], [10]. The parameters are considered as random variables, whose variability is characterized by their continuous or discrete distribution: the value of a specific instance is considered a deterministic sample drawn from the distribution of the parameter. The so-specified parameters with uncertainty are the inputs for the next phase, namely the robust optimization.

❖ **Robust Optimization.** The third phase is the robust optimization process, further detailed in Figure 18. The framework includes two modules: a Model Builder and a Model solver.

The Model builder generates the optimization model based on the deterministic and uncertain parameters; the Model solver processes the model and produces the Pareto-front solutions, which consist in the testing-effort allocation and the assignment of debuggers (and hours) to each system functionality.

The workflow of the Model Solver (here implemented through the NSGA-II algorithm and the MC simulation) is shown in Figure 19.

The algorithm starts with a set of solutions, which represent the initial candidates (i.e., the initial population of the search) - Generating Initial Population step.

At each iteration, recombination and mutation operators are applied to produce ne individuals. The fitness of the solution is evaluated by handling parameters uncertain via MC simulation, with respect to the three objectives, i.e.: i) the expected number of faults that will be detected and corrected by adopting that solution, ii) the testing and debugging cost that will be sustained, and iii) the time to complete the testing activity. The most promising individuals are selected (i.e., Evaluating Individuals in Figure 19) by the metaheuristic. Then, new candidates are generated from the current population (i.e., Generating New Population in Figure 19), until the stop criteria are satisfied4.

Embedding MC simulation within the metaheuristic allows generating robust solutions: the output is not a point solution (where the impact of the input parameters uncertainty on the solution is unknown), but interval, i.e., range of solutions that reflect the possible variability of the optimal solution depending on the variability of the uncertain input parameters. As a result, the tester can select a solution based on more or less conservative criteria (e.g., taking the solution on the lower bound of the 95% confidence interval of the mean of one objective, such as the number of corrected faults).

In the following section, we first describe the MC-based strategy to manage the uncertainty and produce robust solutions. Then, we detail the objective functions of the model and the constraints.

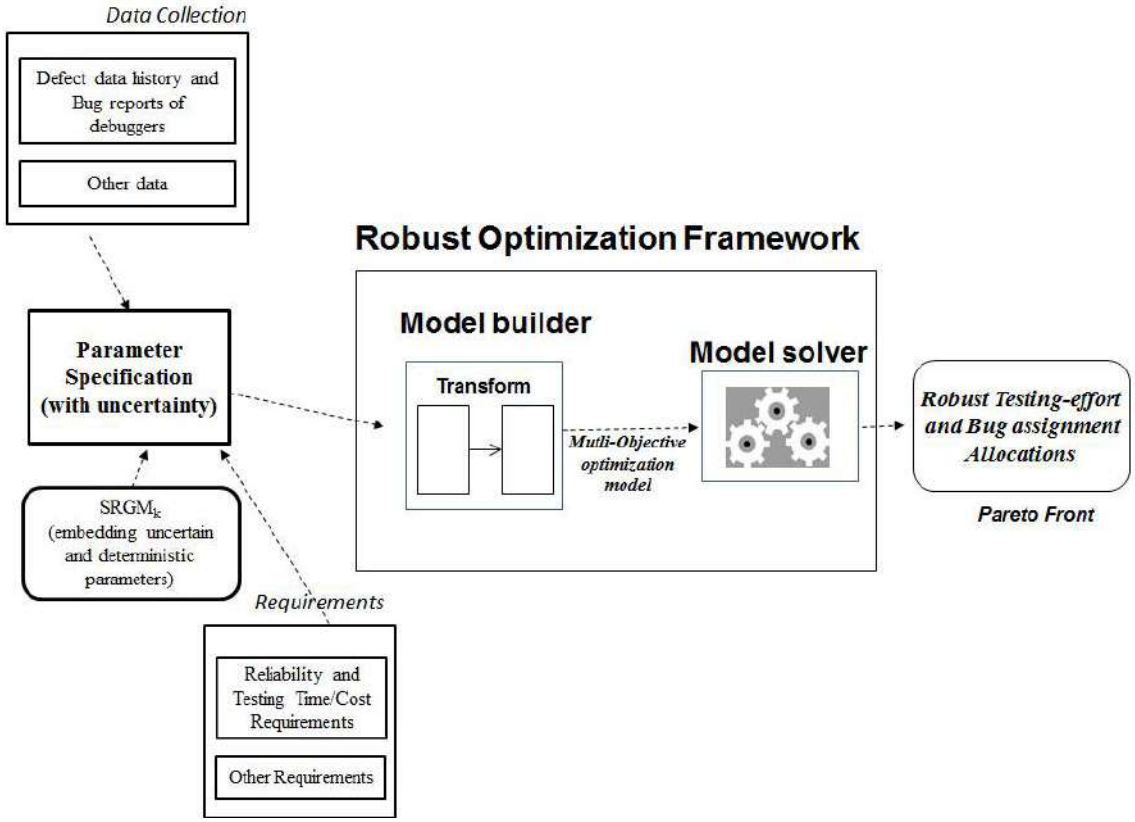


FIGURE 18: THE ROBUST OPTIMIZATION FRAMEWORK AND ITS ENVIRONMENT

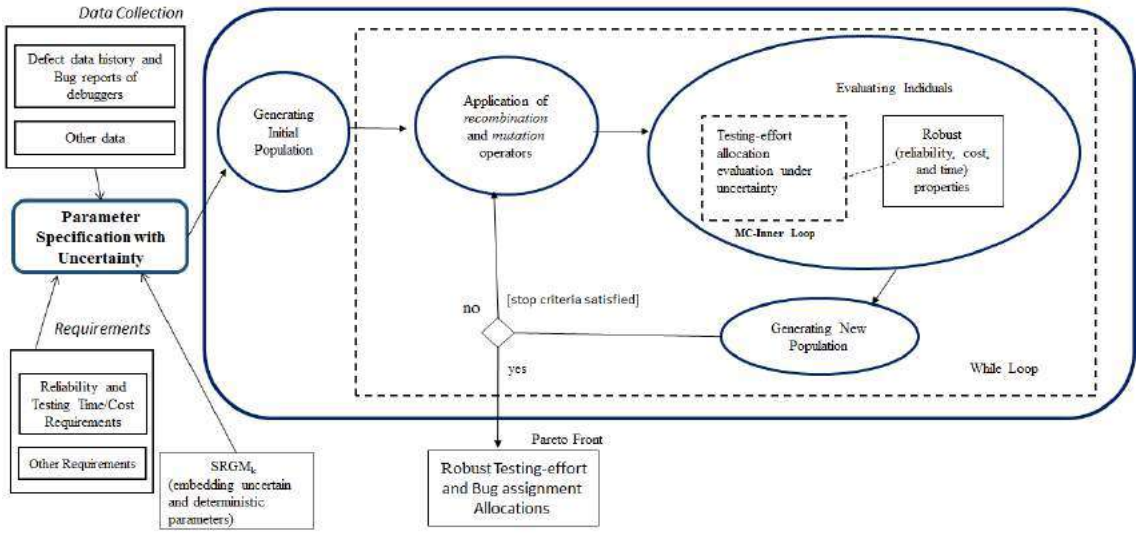


FIGURE 19: HIGH-LEVEL (NSGA-II AND MC-BASED) MODEL SOLVER OVERVIEW

4.2 TESTING-EFFORT ALLOCATION EVALUATION UNDER UNCERTAINTY

In general, system engineering disciplines (and in particular, software testing) are fraught with different types of uncertainties. Software testing, like other development activities (e.g., the design process [15]), is in fact human intensive and thus introduces uncertainties. Software testing uncertainties may affect the development effort and should therefore be accounted for in the test plan [16].

Testing activities are related to the planning and enactment, where enactment includes test selection, test execution, and test result checking. The majority of these activities concern with human behavior (such as test result checking is highly routine and repetitious and thus are likely to be error-prone if done manually [16]). Test enactment is in fact inherently uncertain, since only exhaustive testing in an ideal environment guarantees total confidence in the testing process and its results. However, an ideal testing scenario is infeasible in practice for all but the most trivial software systems. Instead, multiple factors exist that introduce software testing uncertainties [17]. Uncertainty can in fact arise from different sources including external factors not directly related to the behavior of humans in testing activities, such as the usage of the system from end-users.

Different types of uncertainty can thus be faced during the testing process. Example of uncertainty sources is related to the system specification.[4] For example, information on the software system to be tested may be incomplete, such as (some) scenarios, describing the system's dynamics, might not be available (or sufficiently detailed) [18].

The importance and the need of handling uncertainty in software testing is also pointed out by [19]. In particular, the work identifies a set of requirements for adequate uncertainty handling in testing, and outlines the lack of: (i) richer testing frameworks to handle input parameters uncertainty (i.e., specify input distribution instead of discrete inputs), (ii) probabilistic oracles to handle uncertainty associated to the system behavior (i.e., due to misbehaviors and incorrect outputs), and (iii) richer models to deal with system and environment uncertainty.

In this work package, we dealt with the uncertainty affecting the parameters involved in the resource allocation process. The uncertainty is mainly dependent on estimation of the parameters inferred from observed data (e.g., parameters of the SRGMs, average fixing time), or that cannot be accurately evaluated when no enough information is available (e.g., the usage profile of the system functionalities).

We face this problem by combining MC simulation and metaheuristic search in order to assess the robustness of a solution against uncertainty. Our strategy leverages its basics from recent research done in different areas, i.e., software

[4] Notice that this uncertainty source corresponds to the type of uncertainty related to system models, i.e., all sorts of approximation and modeling uncertainties of a design process [15].

architecture quality (e.g., performance and reliability), optimization under uncertainty [109], [110]. Robustness is the ability to tolerate uncertainty in the input parameters. Such as indicated in Figure 19, the search space exploration is achieved by enhancing metaheuristic techniques (the NSGA-II algorithm in particular) with MC simulation for uncertainty analysis. Again, we represent the uncertainty of the parameters by probability distributions to simulate the impact of solution alternatives on objective functions through MC simulations, and to shortlist a set of alternatives using Pareto-based multi-objective optimization techniques.

The approach to evaluate the objective functions in a robust way is depicted by Figure 20. The three objective functions (FCO, TTO, TCO) for a given solution are evaluated by simultaneously considering the uncertainty of all the parameters. The samples are generated based on the probability distribution associated with each uncertain input parameter, and the fitness (as well as the constraints) for the candidate solutions are re-computed for each sample.

Statistical analysis on the fitness values (collected at each MC run) is performed, so as to provide solutions with a desired statistical confidence. In the following pargarpahs, we detail the steps as shown in Figure 20.

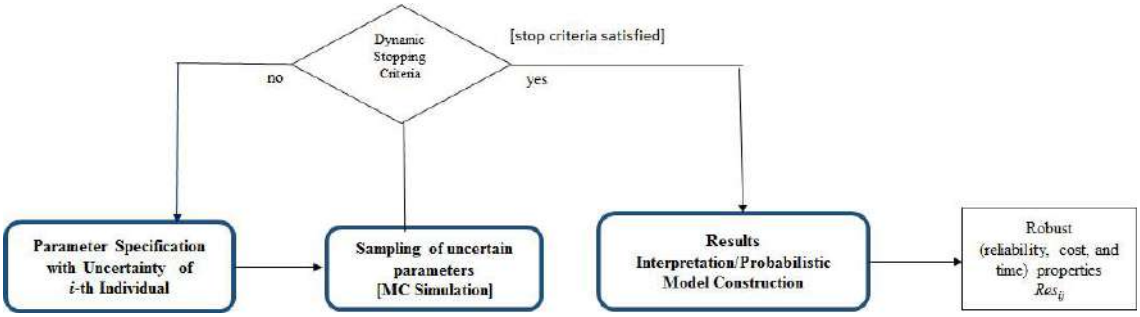


FIGURE 20: *EVALUATION OF TESTING RESOURCE ALLOCATION'S RELIABILITY (TESTING TIME AND COST) UNDER UNCERTAINTY*

4.2.1 Specification of Uncertain Parameters

The uncertain parameters in the testing resource allocation process are categorized as follows::

- *System-specific parameters*. This category includes the parameters related to the detection and correction process, which are dependent on the features of the system (functionalities) under test. These are the parameters of the debug-aware SRGMs of each of functionality, i.e.: (i) the expected number of initial faults; (ii) the parameters of the detection rate per remaining fault function; (iii) the parameters of the correction rate per pending fault function .

- *Parameters specific to the testing-process*. This category includes the parameters related to the testing process and its activities, such as debuggers aspects (e.g., the average amount of bugs a debugger can fix in a day).
- *Usage profile*. The usage profile concerns how users interact with the system. It roughly expresses how much each functionality is expected to be used during operations. When available, this information is exploited at testing time to exercise the system functionalities proportionally to their expected usage. A simple, but widely adopted, way to express the operational profile is the relative (percentage) frequency of invocation of each functionalities (e.g., the call rates of system functionalities).

  Call rate estimates can be usually obtained by examining (i) data gathered during simulation, static profiling, or dynamic profiling; (ii) field data gathered obtained during runtime monitoring of similar systems (or the same system in previous versions); (iii) by exploiting domain knowledge and information provided by the software architecture [20]. It is worth noting that such estimates are affected by uncertainty that we take into account.

Uncertain parameters are treated as random variables. Hence, the values of the parameters are considered as samples of a – continuous or discrete – probability distribution. Distributions of parameters can be derived in several ways [52], such as: (i) using the source of the variations, in the cases when the source of uncertainty is known and can be estimated, (ii) by constructing a histogram, when a considerable amount of data regarding the parameter behavior are available, (iii) approximated as a uniform distribution if no information is available and (iv) as a discrete distribution, when parameters are discrete-valued. Depending on the available information, any of these methods can be selected to derive a sampling distribution for each parameter.

We adopt the uniform distribution (UD) in all the cases but one, as we assume the more general case of no prior knowledge about any parameter. Specifically, the continuous UD over a range is used for the SRGM parameters about fault detection process and for the debugger capacity parameters, while a discrete UD over the set of functionalities is used for the usage profile parameters. For the SRGM parameters of the fault correction process, we exploit the knowledge available from the literature, and adopt the exponential distribution, since it has been shown to well represent the debugging process [111]. In the case of SRGM parameters, the ranges of the uniform distribution can leverage from the confidence intervals (e.g., at 95%) of the parameter estimation (e.g., as in [9][10). For the debugger load capacity, it should be derived from requirements within the organization, which establishes how many (minimum and maximum) bugs each debugger can be assigned in a day. As for the usage profile, if no information is available about which functionality is going to be more used in operation, each functionality can be assigned the same probability. Finally, as for the correction process, the mean of the exponential distribution can be estimated by means of historical data available within a company about the average bug fixing time, as

recorded in a bug tracking system; if the information is not available, a domain expert should assess it.

4.2.2 *Sampling of Uncertain Parameters and Solution Evaluation*

Samples are drawn from the defined distributions for each of the input parameters. These are used in the objective functions and constraints of the model in order to evaluate a candidate solution under the sampled parameter values. The process is repeated until a desired accuracy is achieved (an iteration is called a MC run). The output of a MC run is a sample representing one possible fitness value of the candidate solution (i.e., a triple of values for FCO, TTO and TCO). Criteria for stopping the simulation and robustly evaluate the candidate solution are explained hereafter.

Objective functions evaluation under uncertainty. The robust value for the objective functions from the MC runs could be derived by using two methods [10]. The first method consists in deriving a Probability Density Function (PDF) for each objective function (i.e., a histogram is constructed for each objective by using various discretization techniques), and obtaining the robust objective values for a given confidence. However, this approach is computationally expensive (considering that it should be repeated for all the individuals). Moreover, prospective probability distributions for the objective function values need to be specified a priori.

The alternative method leverages non-parametric or distribution-free statistical procedures. Specifically, for each candidate solution, it derives descriptive statistics (e.g. percentiles, mean, variance or confidence bound) for the three objectives from the observed samples of the MC simulation. To capture the robustness of a candidate with different degree of tolerance, appropriate percentiles can be used as robust objectives. In contrast to the PDF-based method, this method does make any assumption on the probability distribution, being it a non-parametric method, and are successfully applied in a variety of statistical problems.

We hereafter adopt a non-parametric method. Several options are available regarding the descriptive statistic to adopt: for instance, selecting the 50th percentile for all the three objectives means that we consider, for each objective, the median of the observed samples of the MC simulation, for a given candidate solution. A more conservative choice is to select the lower/upper bound, namely the 5th or 95th percentiles, depending on whether the objective is to minimize or maximize. This approximates the bounds of 95% confidence interval. For instance, if the objective is to maximize (such as in the case of FCO), we consider the lower bound as robust solution (namely the 5th percentile of observed values); whereas, for the other two objectives (TTO and TCO), the 95th percentile can be taken as robust solution.

Dynamic Stopping Criteria. Regardless the percentile chosen, the issue of how many MC runs (i.e., how many samples) should be performed for an accurate estimate need to be addressed. We use the notion of dynamic stopping criterion,

introduced in [10], in order to monitor the accuracy of the value to estimate (e.g., number of faults corrected) and automatically stops the MC simulations when the number of samples is sufficient to satisfy a predefined error threshold. For instance, let us consider the objective 1, FCO. Let us denote with *f* a value of this objective after one MC run. Several runs of the MC simulation will provide a set (likely different) values of *f*, due to the different (uncertain) input parameters' values sampled at each run ($F=f_1, f_2, \ldots, f_N$). The goal is to figure out how many samples are needed (i.e., the size of *N*) to get an estimate of the desired percentile of the set *F* – let us denote it as f_{perc}. The procedure is as follows:

- A minimum of *k* MC runs are performed. After *k* repetitions, the desired percentile is estimated on the collected set ($f_1, \ldots, f_k$), obtaining the first estimate of the percentile, f_{perc_1}.
- As the number of runs increases beyond k, further estimates are obtained, considering samples from the beginning, i.e.: $\hat{f}_{prec_2}$ from $\hat{f}_1 \ldots \hat{f}_{k+1}$; $\hat{f}_{prec_3}$ from $\hat{f}_1 \ldots \hat{f}_{k+2}$;, and so on. The variation of the estimate is monitored for a sliding windows of size k, as the accuracy of the estimation is a changing property. Thus, the last k estimates are considered: $\hat{f}_{prec_j}$, $\hat{f}_{prec_{j+1}}$, $\ldots \hat{f}_{prec_{j+k}}$ The statistical significance is calculated for the last k estimates as in [112]:

$$e = \frac{2z_{(1-\frac{\alpha}{2})}}{\sqrt{k}} \frac{\sqrt{\overline{\hat{f}_{prec}^2} - (\overline{\hat{f}_{prec}})^2}}{\overline{\hat{f}_{prec}}} \qquad (1)$$

where *e* is the relative error, $\bar{\hat{f}}$ denotes the average of last k estimates, $\overline{\hat{f}^2}$is the mean-square of the last k estimates, is the desired significance of the test and
z refers to the inverse cumulative density value of the standard normal distribution. The relative error *e* is checked against a predefined tolerance level (0.01 in our case): when it is below the threshold the MC runs are stopped, as the desired accuracy has been achieved.

Robust Optimization. With the MC runs for each candidate solution embedded in the loop, the search space exploration is achieved by enhancing the metaheuristic techniques (the NSGA-II, in our case) with the MC method for the analysis of uncertainty.

For each candidate solution, the fitness value (for each objective) to consider is the chosen percentile (e.g., the 5th, the 50th, or the 95th percentile). The Pareto-front concept is enhanced to express the robustness of the solution with respect to parameters uncertainty. Thus, the dominance notion is slightly modified to account for this change. For instance, suppose we are considering an objective to minimize (e.g., the objective 2, namely TTO). In this case, we may want to consider the upper bound (i.e., 95th percentile of the MC sample set) as conservative criterion to compare solutions. Then, the Pareto-front concept is modified as follows.

Given the minimization of a vector function **f** of n components $f_k, k = 1, \dots, n$ of of a vector variable **x** in D_{om}, subject to inequality and equality constraints $(g_j(x) \geq 0, j = 1, \dots, J$ and $h_k(x) = 0, k = 1, \dots, K)$.
Let us denote with $\bar{f}(x) = (\overline{f_1}(x), \dots, \overline{f_n}(x))$ the upper bound function vector, (where $\bar{f}_i$ is the confidence upper bound of f_i obtained from MC runs). A solution vector $\bar{u} = \{\overline{u_1}, \dots, \overline{u_k}\}$ dominates a vector $\bar{v} = \{\overline{v_1}, \dots, \overline{v_k}\}$, denoted by $\bar{u} \preccurlyeq \bar{v}$ if $\bar{f}(u)$ is partially less than $\bar{f}(v)$, i.e., $\forall i \in \{1, .. k\}, \ \bar{f}(u)_i \leq \bar{f}_i(v) \ \wedge \ \exists i \in \{1, \dots, k\} : \ \bar{f}(u)_i < \bar{f}_i(v)$.

Project Constraints evaluation under uncertainty. Figure 7 sketches a high level view of the proposed approach for evaluating alternative candidates (i.e., testing resource allocation individuals, see Figure 19) according to the constraints on reliability (and testing time/cost).

The input of the approach for constraints evaluation is a testing-effort and bug assignment allocation (an individual of the population of the search). It proceeds iteratively. At each iteration step, the individual is evaluated according to the constraints on reliability/time/cost of testing (see Figure 20). Such properties (i.e., reliability, cost and time of testing) of one individual are evaluated by simultaneously considering all the parameters' uncertainties. In particular, samples are generated from the probability distributions of uncertain parameters using the MC method, and the properties are re-calculated for each of these samples. The output of the constraints evaluation approach, Res_{ij} (with j representing the property identifier), is a descriptive statistic (e.g. percentile, mean, variance or confidence bound) for the properties (reliability, testing time and cost) from the observed samples of the MC simulation. Dynamic stopping criteria are used for determining when a sufficient number of samples for the associated individual is determined.

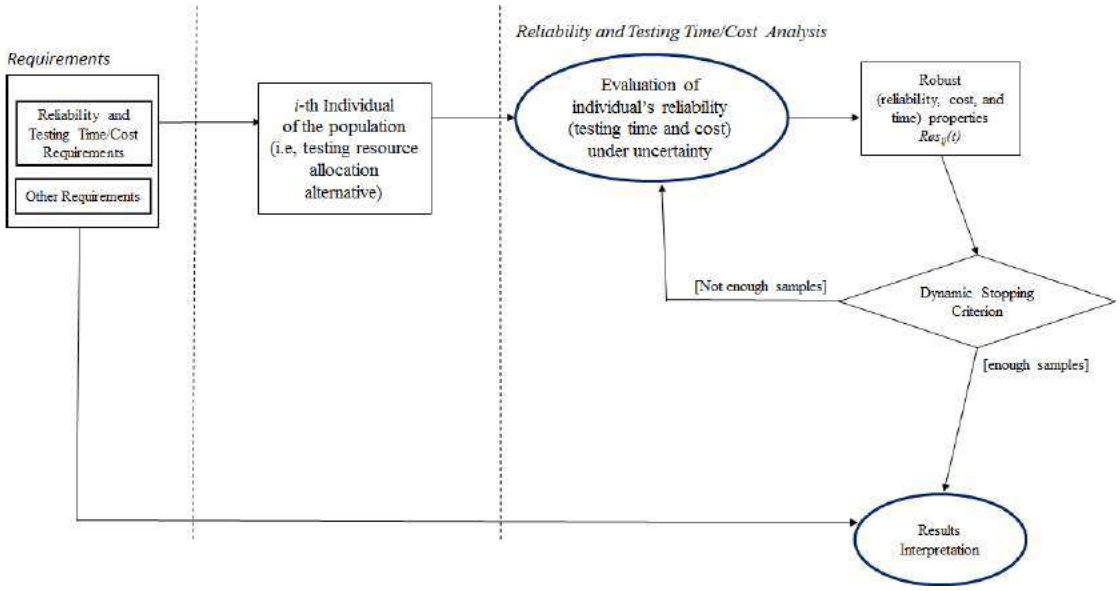


FIGURE 21: CONSTRAINTS ON RELIABILITY AND TESTING TIME/COST EVALUATION PROCESS IN PRESENCE OF UNCERTAIN PARAMETERS

Stopping Criteria **(**Figure 21**).** We have defined the stopping criteria by exploiting the work in [9] that deals with the model-based performance analysis (i.e., the satisfaction of certain performance requirements, e.g. response time, throughput) of software architectures under uncertain parameters. The work introduced a MC-based approach. In particular, the sampling process is seen as a Bernoulli experiment where each trial provides a value of 1 or 0 leading to a Bernoulli distribution with parameter p (which can be estimated using MC simulations). Stopping criterion has been defined for estimating the value of p with a tolerance against the inherited uncertainty.

Similarly, we can consider the MC-based evaluation process of constraints (illustrated in Figure 21), as a Bernoulli experiment where each trial (corresponding to execution of the evaluation process, see Figure 20) provides a value of 1 or 0 leading to a Bernoulli distribution with parameter p, i.e., each execution of the evaluation process has one Boolean indicator representing whether the trial satisfies reliability (cost and time) requirements. In other words, a run of our constraint evaluation process corresponds to a sample of the MC-based process defined in [9].

Thus, the stopping criteria can be defined (by exploiting the ones used in [9]) as follows:

– A minimum of h executions of the MC-based process (of Figure 20) are conducted and results are recorded $(x_1,\ldots,x_h)$. The value of p is estimated as follows:

$$\hat{p} = \frac{\sum_{i=1}^{h} x_i}{h} \qquad (2)$$

- The variation of the estimate $\hat{P} = \widehat{p_1}, \widehat{p_2}, \ldots \widehat{p_h}$ is monitored for a sliding window of size h. Only the last h executions of the MC-based process are monitored, as the accuracy of the estimation is a changing property. The objective is to detect if sufficient accuracy is obtained.
- The statistical significance is calculated for the last h estimates:

$$e = \frac{2z_{(1-\frac{\alpha}{2})}}{\sqrt{h}} \frac{\sqrt{\overline{\hat{p}^2} - (\bar{\hat{p}})^2}}{\bar{\hat{p}}} \qquad (3)$$

where e is the relative error, $\hat{p}$ is the average of last h estimates, $\hat{p}^2$ is the mean square of the last h estimates, α is the desired significance of the test and z refers to the inverse cumulative density value of the standard normal distribution. The relative error e of the reliability (cost/time) estimate $\hat{P}$ is checked against a tolerance level, e.g. 0.005.

Results Interpretation **(**Figure 21**).** Similar to the performance robustness of software architectural models [9], the robustness of testing resource allocations with respect to the requirements on reliability (and testing cost and time) can be evaluated by systematically analyzing the results, $Res_{ij}(t)$ (with j and t

representing the property identifier and the run identifier, respectively) of the MC-based evaluation process runs, and checking if each evaluation process's run fulfills the constraints.

We associate to the *t*-th result, $Res_{ij}(t)$, corresponding to the *t*-th run of the constraint evaluation process, a fulfillment flag $f_{Res_{ij}}(t)$ which is a binary value that indicate the satisfaction of the requirements. The robustness of the testing resource allocation (corresponding to the *i*-th individual) with respect to the requirements on reliability (testing time and cost) is defined as follows:

$$robust_{f_{Res_{i,j}(t)}} = \sum_{t=1}^{N} \frac{f_{Res_{i,j}(t)}}{N} \qquad (4)$$

where (i) $robust_{f_{Res_{ij}(t)}}$ is a real value in the [0,1] interval, and (ii) *N* is the number of execution of the constraint evaluation process. It is the percentage of samples that fulfill the requirement(s).

4.3 OPTIMIZATION MODEL FORMULATION

The goal of our optimization model is to find the optimal allocation of testing resources among *K* functionalities of a system *S* to test, and optimal assignment of bugs to debuggers to maximize the effectiveness of the testing process. "Optimal" here denotes actions that incur minimum time and cost of testing, and maximum effectiveness of the fault correction process under minimum reliability and testing budget constraints.

Table 15 summarizes the symbols used throughout this section. e Section. In the following, the parameters, variables, constraints, and objective functions are described.

TABLE 15: MAIN NOTATION ADOPTED

Symbol	Description
$\mathcal{K}$	Number of system functionalities
k	System functionality index
a_k	Expected number of initial faults in the functionality k
ω_k	Probability that the functionality k will be invoked
δ_k	Average number of hours required for fixing a bug of the functionality k
R	Minimum threshold given to the reliability on demand of the system
t_0	Time at which the resource allocation model is run
Y_0	Testing effort (measured in man-hours) already spent at time t_0 (in the case of "dynamic" allocation, i.e., during testing – see Section III – it is different from 0)
$F_{d\&c}(t_0)_k$	Number of faults (of the functionality k) already detected and corrected at the t_0
$\mathcal{D}$	Total number of debuggers
d	Debugger index
γ_d^k	Average number of hours in a day that the debugger d can work to fix bugs of the functionality k (# of hours over 24h)
x_d^k	Debugger d used/not used to fix bugs of the functionality k (1/0, respectively)
N_d^k	Time assigned to the debugger d for testing the k-th functionality (hours)
$y_k(t)$	Instantaneous testing-effort at time t for the functionality k, estimated by a generalized *logistic* testing-effort function (man-hours)
$Y_k(t)$	Cumulative testing effort devoted to functionality k in $(0, t]$ (man-hours)
t_k	Calendar testing time devoted to test the functionality k (hours)
$m_{d_k}(t_0 + t_k)$	Expected cumulative number of faults detected in functionality k at testing time $t_0 + t_k$
$m_{c_k}(t_0 + t_k)$	Expected cumulative number of faults corrected in functionality k at testing time $t_0 + t_k$
$\lambda_k(t)$	Fault detection rate per undetected fault for the functionality k
$\mu_k(t)$	Fault correction rate per detected but uncorrected fault for the functionality k
α	Consumption rate of testing-effort expenditures in the logistic testing-effort function (TEF)
h	Structuring index in the logistic TEF whose value is larger for better structured software development efforts
$\mathcal{B}$	Total amount of testing-effort available for consumption (man-hours)
$\mathcal{A}$	Constant parameter in the logistic TEF
C_1^*	Average cost per man-day to correct a bug during testing
C_2^*	Average cost per man-day to correct a bug in operational use
C_3^*	Average cost of testing a functionality per unit testing-effort expenditure, expressed in cost of a man-day
$\phi_k(t)$	Expected failure intensity function for functionality k at testing time t
ϕ^*	Maximum threshold given to failure intensity of the system after test

4.3.1 Model Parameters

In this section, we describe the main parameters of our optimization model.[5]

In this section, we describe the main parameters given as input to the model:

– The time, t0 is the time at which tester decides to run the resource allocation algorithm. This time can be the beginning of the testing process of the system under test (when historical data are used for the SRGMs construction) or it can be any time during the testing process (when online testing data are used to build the SRGMs). In the latter case, the allocation model can be run several times during testing (what we called dynamic allocation); thus we refer to t_0 as "(re-)iteration" time.

– $F_{d\&c}(t_0)k$ is the number of faults detected and corrected in functionality k after t_0 time units.

– When the algorithm has to be run, the SRGMs for each functionality should be available, according to the phase 1 of the approach. They are characterized by detection and correction rate functions, denoted as $\lambda_k(t)$, and μ_k(t), representing, respectively, the fault detection rate per undetected fault, and the fault correction rate per detected but uncorrected fault. Their parameters' estimation can be coped with in several ways (e.g., Maximum Likelihood Estimate, Least Square Estimate, or Expectation Minimization).

[5] For the sake of readability, other parameters are given later in the document.

– The δ_k parameter is the average number of hours required to fix a bug, for the functionality *k*. It is estimated by querying historical data about bug correction tracked in the bug repository[6], such as in [58],, [113], taking the median instead of the mean when the distribution is highly skewed.

ω_k is the probability that the k-th functionality will be invoked:
$w_k \geq 0, \forall k = 1, \ldots K$, and $\sum_{k=1}^{K} w_k = 1$. This information can be synthesized from the operational profile estimation [38], according to either design-time (e.g., documentation, simulation, profiling) or execution-time (field data of previous versions) methods, possibly complimented by expert judgment [114].
– γ_k^d is processing capacity of the debugger d with respect to the functionality *k*. It represents the working rate of the debugger on functionality *k*, expressed as average number of hours per day that the debugger *d* is allowed to work *t* fix bugs of functionality *k*.

- C_1^*, C_2^*, C_3^* are the cost parameters used in the cost-related objective function (TCO). They represent, respectively: (i) C_1^* is the cost per man-day to correct a bug during testing; (ii) C_2^* is the cost per man-day to correct a bug during operational use (typically $C_2^* > C_1^*$ [7]); and (iii) C_3^* is the cost per testing-effort expenditure unit (e.g., man-hour or man-day) to test a functionality (i.e., hourly or daily cost of a tester). These parameters are provided as input by the user; although they could generally have different values for each functionality, we assume they are the same for each functionality to keep the model simple.

1) α, h, β, A are the parameters of the logistic testing effort function (TEF) [30][26], which is used to explain how testing effort varies in function of calendar time. Specifically: α, is the consumption rate of testing-effort expenditures, (ii) h is a structuring index whose value is larger for better structured software development efforts, (iii) β is the maximum budget that has been given on the total amount nof testing-effort that can be consumed (expressed in man-hours), and (iv) A is a constant parameter. Although the estimate of these parameters is not the main focus of our work, as shown in [26], [25], and [24], [22], they may be estimated by the method of least squares (LSE) or maximum likelihood estimation (MLE).

4.3.2 Variables

This section introduces the decision variables of the optimization model.

The Y_k $(1 \leq k \leq K)$ variables represent the amount of testing effort (in man-hours) to perform on each system functionality. It is a decision variable, namely: solving the model will provide a vector of Y_k values, that are the suggested testing efforts to spent per functionality. A related variable is t_k: it is the calendar testing time (measured, in hours or in days) devoted to test functionality *k*, and is bound

[6] For simplicity, we assume the average number of hours required to fix a bug of a given functionality k (i.e., δk) is the same for each debugger d working on that functionality.

to the spent testing effort Y_k via the TEF: In fact, as the effort is related to testing time by the TEF, assigning Y_k man-hours to k corresponds to assign $t_k = F^{-1}(Y_k)$ hours, where F^{-1} is the inverse of the TEF.

The x_d^k $(1 \le d \le D, 1 \le k \le K)$ and N_d^k $(1 \le d \le D, 1 \le k \le K)$ variables are used to predict the correction process of the debugger/tester d with respect to the functionality k. These are further decision variables. One of the goals of the model is, in fact, to maximize the number of faults corrected, which is related not only to the maximization of faults detected, but to how much effectively such revealed faults are corrected by debuggers. Specifically, the x_d^k variables are used to select debuggers for the functionality k; in particular x_k^d is equal to 1 if the debugger d is chosen and 0 otherwise. The N_k^d variables represent the time (in hours) assigned to the debugger d to work on functionality k in the interval $(t_0,t_k]$.

Thus, a solution consists of: the Y_k variables $(1 \le k \le K)$ suggesting the optimal testing effort per functionality, by the x_d^k $(1 \le d \le D, 1 \le k \le K)$ variables and, assigning debuggers to functionality, and by the N_d^k $(1 \le d \le D, 1 \le k \le K)$ variables assigning the number of hours of debuggers to functionalities.

4.3.3 Constraints

A first set of most relevant constraints of the model are expressed in Figure 22:

1. $\sum_{d=1}^{\mathcal{D}} N_d^k \ge \delta_k(m_{d_k}(t_0 + t_k) - m_{d_k}(t_0)), \quad \forall k = 1 \dots \mathcal{K}$
2. $N_d^k \le \frac{t_k}{\gamma_d^k} \cdot x_d^k, \quad \forall k = 1 \dots \mathcal{K}, \forall d = 1 \dots \mathcal{D}$
3. $\begin{cases} x_d^k = 1 & \text{iff} \quad \text{Debugger } d \text{ must be assigned to } k; \ k \in \{1 \dots \mathcal{K}\}, d \in \{1 \dots \mathcal{D}\} \\ x_d^k = 0 & \text{iff} \quad \text{Debugger } d \text{ must not be assigned to } k; \ k \in \{1 \dots \mathcal{K}\}, d \in \{1 \dots \mathcal{D}\} \end{cases}$
4. $m_{d_k}(t_0 + t_k) - m_{d_k}(t_0) \le a_k - F_{d\&c}(t_0)_k, \quad \forall k = 1 \dots \mathcal{K}$
5. $\sum_{k=1}^{\mathcal{K}} Y_k \le \mathcal{B}, \quad \forall k = 1 \dots \mathcal{K}$
6. $Y_k \le \mathcal{B}(1 - \prod_{d=1}^{\mathcal{D}}(1 - x_d^k)), \quad \forall k = 1 \dots \mathcal{K}$
7. $\sum_{k=1}^{\mathcal{K}} \omega_k \cdot \phi_k(t_k) \le \phi^*$

FIGURE 22: MODEL **CONSTRAINTS**

- For each functionality k, faults detected in the interval $(t_0,t_k]$ must be fixed. Equation 1 in Figure 22 expresses that the total time assigned to debuggers on functionality k must be greater or equal than the expected time to correct the detected bugs (estimated as mean fixing time per bug multiplied by the expected number of bugs that will be detected if k is tested for a time t_k). Note that this equation holds if we assume that all

detected bugs with the allocated testing resources must be corrected, and thus assigned to debuggers, whereas the equation should be appropriately modified if this assumption is relaxed.

- The bug correction process is modeled as a function of the amount of time (e.g., in hours) required to fix the bugs detected, and as function of the working time of debuggers. The waiting queues are modeled by introducing a constraint on the capacity of debuggers. This constraint is expressed by Equation 2 in Figure 22: for each functionality *k*, the load of debugger *d* due to the assignment of bugs is limited by a function of the processing capacity of debugger d, (i.e., γ_k^d). N_k^d is greater than 0 only if: i) the debugger *d* is allocated to functionality *k* ($x_d^k = 1$), ii) a non-zero testing time t_k is allocated functionality *k* ($t_k > 0$), and, from constraint 1, iii) at least one bug is expected to be detected during the assigned time t_k (i.e., $m_{d_k}(t_0 + t_k) > m_{d_k}(t_0)$), assuming $\gamma_k^d > 0$ and $\delta_k > 0$. This throughput model is a light-weighted one that favors model solvability. An explicit management of queues could be introduced, using, for example, queuing network models explicitly considering a one-to-one mapping between debuggers and bugs, but at the expense of computational complexity and understandability.
- Equation 3 of Figure 22 indicates the (possible) constraints defined for debuggers that must be assigned or cannot be assigned to functionalities for some reasons, e.g., due to the debugger's skill level or expertise area. In these cases, the corresponding variable x_d^k is forced to be 1 or 0. Note that, in order to solve incompatibilities or dependencies among debuggers and/or functionalities, due, for instance, to human factors (skill set, skill level and availability) or functionality characteristics, additional constraints can be added as contingent decisions. For example, $x_2^1 \leq x_3^2$ means that, if the second debugger is selected for the first functionality, then the third debugger must be selected for the second functionality; $x_1^2 \leq x_1^3$ means that, if debugger 1 is selected for functionality 2, then he must also be selected for functionality 3.
- Equation 4 in Figure 22 states that the expected number of cumulative faults detected in $(t_0, t_k]$ (namely, if $t_k = F^{-1}(Y_k)$ testing time is assigned to test *k*), cannot be greater than the expected number of residual fault in *k*.
- Equation 5 in Figure 22 expresses a constraint on the maximum effort that can be allocated. A maximum threshold *B* is given on the total amount of testing effort possibly consumed (expressed in man-hours). The test manager has to distribute a budget *B* of man-hours among the *K* functionalities; the solution suggests that *k*-th functionality should receive a testing effort equal to Y_k man-hours.
- Finally, Equation 6 of Figure 22 tells that: if there are no available debuggers for functionality *k*, then the amount of testing effort allocated to *k* (i.e., Y_k) will be 0 (since bugs could be detected, but then not corrected). In other words, if the functionality *k* receives a certain amount of testing effort, then one or more debuggers must be assigned to functionality *k*. There could be an additional constraint on Y_k: if we require that all the functionalities must be tested, then $Y_k > 0$, $1 \leq k \leq K$. Similarly, further requirements by the tester could be seamlessly included as constraints in

the model, enabling several extensions; in this work, we keep the model in its basic form.

- Equation 7 reports the constraint on the minimum desired failure intensity at the end of testing. The estimate of failure intensity of a functionality k is usually obtained though the SRGM, as it is the derivative of the cumulative expected number of detected faults, $m_d(t)$. The estimate is obtained as:

$$\phi_k(t_k) = \frac{dm_d}{dt}(t_k) \qquad (5)$$

It denotes the expected failure intensity if the model solution assigns a testing effort Y_k to functionality k such that $Y_k = F(t_k)$ (where F denotes the TEF), or, similarly, such that: $t_k = F^{-1}(Y_k)$. A maximum threshold, ϕ^*, is given to the failure intensity of the overall system as input requirement. In an average-case scenario, like the one we assume, the failure intensity constraint is formulated as follows:

$$\sum_{k=1}^{K} \omega_k \phi_k(t_k) \leq \phi^* \qquad (6)$$

In other words, the system failure intensity is weighted by the call rates of each functionality. In a worst-case scenario, tester may want to require that all functionalities should satisfy a failure intensity constraint. In this case, the constraint would be formulated as follows:

$$max_{k=1\ldots K}(\phi_k(t_k)) \leq \phi^* \qquad (7)$$

Finer-grained constraint can be introduced to guarantee threshold limits for each functionality, i.e.: $\omega_k \phi_k(t_k) \leq \phi^*$.

4.3.4 Multi-Objective Function

In this section, we define the three objectives of the multi-objective optimization problem.

2) **Fault correction process' Effectiveness Objective (FCO)** The objective function to be maximized, as the predicted number of faults corrected (providing an assessment of the system reliability after the application of the amount of testing effort, Y_k, on each of system functionalities), is given by:

$$FCO = \sum_{k=1}^{K} m_{c_k}(t_0 + t_k) \qquad (8)$$

The solution for the exponential case with logistic TEF is:

$$m_{c_k}(t_0 + t_k) = e^{-\mu_k(t_0+t_k)} \int_{t_0}^{t_0+t_k} a_k \mu_k e^{\mu_k s}\left(1 - \exp\left[-\beta_k\left(Y_k(t) - Y_k(0)\right)\right]\right)ds$$

(9)

After the application of the amount of testing effort, Y_k, the expected number of faults corrected for the functionality k depends on: the fault detection rate, related to the testing effort suggested for k, Y_k, through the TEF, and (ii) on the availability of sufficient debugger (hours), regulated by N_d^k and x_d^k variables, for the correction of all detected faults at the rate expressed by $\mu_k(t)$.

3) Testing Time Objective (TTO)

Assuming that the time-depending behavior of the testing-effort (for each of the system functionalities) is modeled by the generalized logistic testing-effort function proposed in [26][30], we can compute the testing time for functionality k can as function of the effort:

$$t_k = \left(-\frac{1}{\alpha * h} ln \left(\frac{(\frac{B}{Y_k})^h - 1}{A} \right) \right) \qquad (10)$$

where (i) α is the consumption rate of testing-effort expenditures in the logistic testing-effort function, (ii) h is a structuring index whose value is larger for better structured software development efforts, (iii) B is a maximum threshold that has been given on the total amount of testing-effort that can be consumed (expressed in man-hours), and (iv) A is a constant parameter in the logistic testing-effort function. Although the estimate of these parameters is not the main focus of our work, as shown in [26], [27], and [28], they may be estimated by the method of least squares. Moreover, more details on estimation of the budget B can be also found in [5].

Assuming that manpower is available to independently test system functionalities (namely, they can proceed in parallel), the second objective function is the time minimization for testing the K functionalities:

$$TTO = min_{k=1\ldots K}(t_k) \qquad (11)$$

Although this assumption could not be too realistic due to the overhead that likely incurs when a lot of functionalities must be tested, it reflects a common practice in testing planning. However, as previously discussed to relax such an assumption, guidelines of existing approaches for the work packages scheduling and staff assignment problem plan could be exploited.

3) ***Testing-effort Cost Objective (TCO).*** The third objective cares about minimization of cost, which is a measure related to the effort spent but that goes beyond the mere effort for testing. In agreement with [30], for the functionality k, the cost of testing effort expenditures during software development and testing phase, and the cost of correcting errors before and after release, can be expressed as follows:

$$Cost_k(t) = C_1^*\left(\frac{\delta_k}{24}\right) m_{c_k}(t) + C_2^*\left(\frac{\delta_k}{24}\right)\left(m_{d_k}(\infty) - m_{c_k}(t)\right) + C_3^*\left(\frac{Y_k}{24}\right) dt \qquad (12)$$

where: (i) $C_1^*\left(\frac{\delta_k}{24}\right)$ is the cost per day to correct a bug during testing; (ii) $C_2^*\left(\frac{\delta_k}{24}\right)$ is the cost of correcting a bug in operational use (typically $C_2^* > C_1^*$ [31]); and (iii) C_3^* is the cost of testing per unit testing-effort expenditure, expressed in cost of a man-day (for a tester).[7]

This cost model, similar to the one in [30], is a light-weighted one that favors model solvability. However, it can be enhanced by using well-assessed cost model from the literature (e.g., COCOMO II model [32]) to increase the result accuracy. This can be done without essentially changing the overall model structure, but with the side effect of increasing the solution complexity. To address this, the guidelines of the COCOMO II-based model defined in [33] for estimating the development cost of an in-house developed service may be exploited. More specifically, in [33], the development cost of an elementary software service has been defined as a function of the testing activity (e.g., the number of tests performed on a service before delivery). The original COCOMO II model [32] 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. These two attributes allows estimating the amount of effort, in terms of person-months, needed to deliver the software.

C_1^* , C_2^* and C_3^* may be estimated in different ways depending on the functionality type and debugger/tester profile. Details on their estimation can be found in [30]. The work in [30] is focused on cost of software modules, whereas we consider the cost to test system functionalities. In other words, we consider each of the system functionalities as software modules.

Therefore, the objective function to be minimized, as the sum of the cumulative testing-effort costs for all of system functionalities, is given by:

$$TCO = \sum_{k=1}^{K} Cost_k(Y_k) \qquad (13)$$

4.3.5 Model Summary

Figure 10 summarizes the formulation of our optimization model.

[7] Notice that the cost C_3^* does not include the costs for the bug-fixing activity. Instead, these costs are considered in the estimation of the C_{1k}^* parameter.

$$\min(-FCO, TTO, TCO)$$

s.t.

$$\sum_{d=1}^{\mathcal{D}} N_d^k \geq \delta_k(m_{d_k}(t_0+t_k) - m_{d_k}(t_0)), \quad \forall k = 1 \ldots \mathcal{K}$$

$$N_d^k \leq \frac{t_k}{\gamma_d^k} \cdot x_d^k, \quad \forall k = 1 \ldots \mathcal{K}, \forall d = 1 \ldots \mathcal{D}$$

$$\begin{cases} x_d^k = 1 & \text{iff} \quad \text{Debugger } d \text{ must be assigned to } k; \ k \in \{1 \ldots \mathcal{K}\}, d \in \{1 \ldots \mathcal{D}\} \\ x_d^k = 0 & \text{iff} \quad \text{Debugger } d \text{ must not be assigned to } k; \ k \in \{1 \ldots \mathcal{K}\}, d \in \{1 \ldots \mathcal{D}\} \end{cases}$$

$$m_{d_k}(t_0+t_k) - m_{d_k}(t_0) \leq a_k - F_{d\&c}(t_0)_k, \quad \forall k = 1 \ldots \mathcal{K}$$

$$\sum_{k=1}^{\mathcal{K}} Y_k \leq \mathcal{B}, \quad \forall k = 1 \ldots \mathcal{K}$$

$$Y_k \leq \mathcal{B}(1 - \prod_{d=1}^{\mathcal{D}}(1 - x_d^k)), \quad \forall k = 1 \ldots \mathcal{K}$$

$$\sum_{k=1}^{\mathcal{K}} \omega_k \cdot \phi_k(t_k) \leq \phi^*$$

Bounds:

$$x_k^d \in \{0,1\}; \ \gamma_d^k \geq 0; \ \delta_k \geq 0; \ t_0 \geq 0; \ F_{d\&c}(t_0)_k \geq 0; \ \omega_k \geq 0; \ \sum_k \omega_k = 1;$$

$$C_1^* \geq 0; \ C_2^* \geq 0; \ C_3^* \geq 0; \ \alpha \geq 0; \ h \geq 0; \ \mathcal{B} \geq 0; \mathcal{A} \geq 0; \ Y_k \geq 0; \ N_d^k \geq 0$$

FIGURE 23: OPTIMIZATION MODEL FORMULATION

Main assumptions and threats to validity

The usage of SRGMs (with TEF) to model the fault detection and correction process implies the following assumptions:

- The fault removal process is modeled as a Non-Homogeneous Poisson process (NHPP), where the mean number of faults detected in the time interval (t, $t + \Delta t$) by the current testing-effort is proportional to the mean number of remaining faults in the system at time t.
- Each of the system functionalities are subject to failures at random times (with independent inter-failure times) caused by the manifestation of remaining faults in the functionalities.
- System functionalities are autonomous, independently testable. New functionalities or feature enhancement are not introduced into the code during testing.
- The relation between testing effort and testing time can be modeled by a testing effort function (TEF).
- – Each time a failure occurs, the fault that caused it is correctly removed and no new faults are introduced (i.e., perfect repair). This assumption can be partially relaxed if we admit, among the set of selectable SRGMs, the ones modeling the imperfect debugging phenomena.

We mitigate the SRGM assumptions by enabling, in the formulation, a module-tailored selection of the best model among a set SRGMs, and by the possibility, in the process, to fit SRGM with online data (that account for the effect of such assumptions' violations). In addition to the SRGM assumptions, further assumptions are:

- We assume historical information about issue reports is correct: namely, reporters can correctly distinguish a bug from a feature request, can correctly identify duplicate bug reports, and we can faithfully approximate the average bug fixing time (e.g., the δ parameter) as the bug closing minus the bug opening time.
- Bug fixing time dependence on other basic bug-related features, such as severity and priority or bug owners and bug types is not considered to keep the model simple at this stage. Extensions can be implemented for more accurate but expensive model formulations.
- We assume that (i) debugger manpower is available to independently fix bugs in system functionalities, and (ii) for each of the Y_k man-hours, there is the same pool of D debuggers. We are working toward relaxing such assumptions. To this extent, we are investigating how to use the guidelines of existing approaches (such as the ones of [35]) for the work packages scheduling and staff assignment problem plan (i.e., the allocation of staff to teams and the allocation of teams to work packages).
- Although we admit several testing-effort time model, we taken, as specific example, the generalized logistic testing-effort function, a widely-used one. It can be replaced by other well-assessed distribution function from the literature. Although this can be done without changing the model structure, the effect of other TEFs on solution complexity are not assessed.
- Cost constants are assumed to be known within the company. Such information is not always easily accessible, and more or less complex models can be adopted to accurately estimate it, as COCOMO ones. Such models are out of scope for this paper.

4.4 HEALTH CARE CASE STUDY

In this section we describe the case study that we devised in order to validate the effectiveness of the approach in dynamic testing resource allocation of industrial health care software. In particular, we present the achieved results on the Medical Company scenario (see Deliverable D2.2 [95]) .

The goal of our experimentation is to evaluate the effectiveness of our approach in addressing the important challenges related to the tradeoff between (i) allocating resources to system functionalities where testing will have the highest detection power, and (ii) maximizing the number of bugs that can be fixed in available time. To do this, we compared the amount of testing efforts selected by our approach with the amount of testing efforts selected without explicitly incorporating bug assignment activities into the fault correction process of each of the functionalities.

Random generation of model instances. Starting from the nominal values of the parameters, we have generated 4 different instances (here also called *perturbed configurations*) by randomly changing the following parameters: (i) the total amount of testing-effort eventually consumed, B; (ii) the average number of hours required for fixing a bug of the functionality k, δ_k , and the expected number of initial faults in the functionality k, a_k. Specifically, the perturbed configuration parameters have been varied within 10% of the nominal values, with the exception of the total amount of testing-effort, B, that has randomly increased of the 10% of the nominal value.

We have applied on the same case study, our approach and the typical state-of-the art testing resource allocation approach (e.g., [7], [3]). Our approach is mainly focused on system functionalities (which we consider as software modules). Therefore, our model can be compared with existing works by introducing a mapping of software modules on system functionalities.

The state-of-art problem of testing resource allocation (here also called *base model*) typically consists of finding the amount of testing-effort to be performed for each of the system functionalities[8] that minimizes the total cost under the threshold R on the system reliability. Additional decision variables are introduced in our optimization model to represent in bug-fixing activities to perform for each system functionalities.

For the experiments, we have used JMetal [37], an object-oriented Java-based framework aimed at the development, experimentation, and study of metaheuristics for solving multi-objective optimization problems.[9] Due to the stochastic nature of evolutionary algorithms, we have performed 30 independent runs per algorithm (see [36] for details).

Our comparison between the two approaches can be summarized in three steps.

Step 0: Let us assume that all the debuggers may work four hours a day for each of the system functionalities.

For each perturbed configuration (and for the nominal instance), we have solved two models for R that spans from 0.9 to 0.97. In Figure 24, Figure 25, and Figure 26, we report the obtained results, where each bar indicates, respectively, the number of corrected faults, the testing time and cost of a model averaged over its four perturbed configurations and nominal instance. Each group of tree bars - corresponding to one model - refers to the model's results with five instances. In particular, each bar - corresponding to the model solution over the four perturbed configurations and the nominal one with a fixed value of the threshold R - reports

[8] As remarked above, for sake of comparison, we introduce a one-to-one mapping of system functionalities on software modules.
[9] jMetal can be obtained freely from http://jmetal.sourceforge.net/.

the highest, lowest and average number of corrected faults, the testing time and cost obtained.

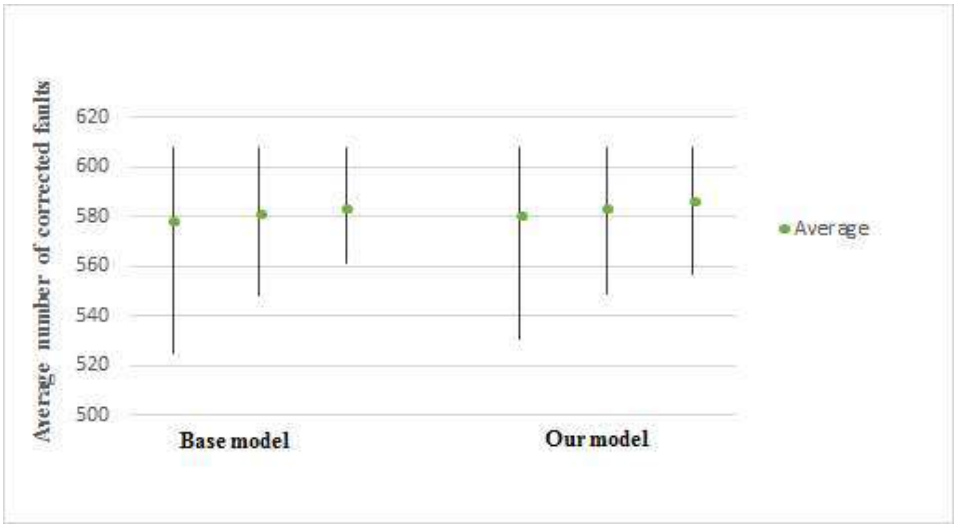


FIGURE 24: AVERAGE NUMBER OF CORRECTED FAULTS VS RELIABILITY THRESHOLDS

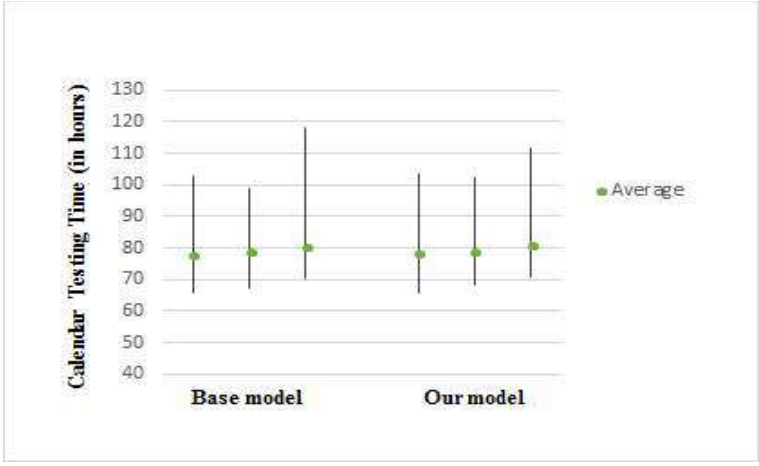


FIGURE 25: CALENDAR TESTING TIME VS RELIABILITY THRESHOLDS

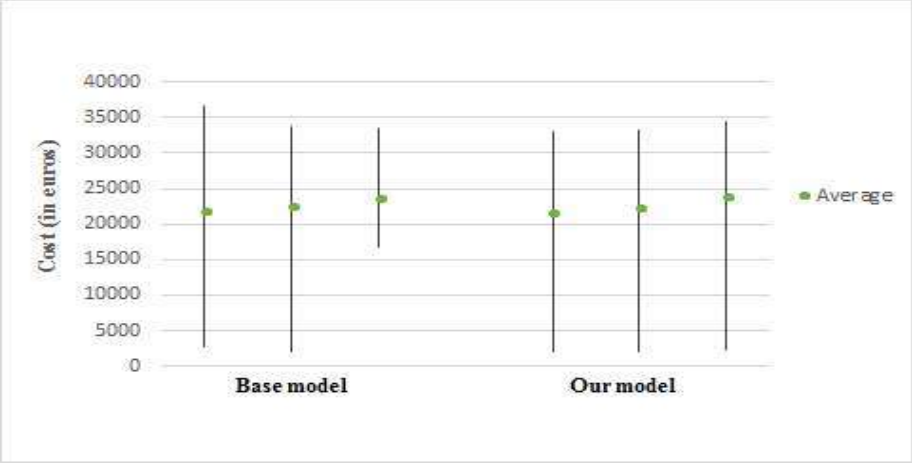


FIGURE 26: COST VS RELIABILITY THRESHOLDS

The results highlight, in general, that the solutions of the *base model* and *our model* do not show discrepancies in case of non-complex search space (i.e., for simple scheduling of debuggers to fix-activities), in that the average number of bugs, times and costs of their solutions are only slightly different. Moreover, for a given model, the times and costs slightly increases while increasing the reliability threshold *R*. This can be observed by fixing a value on the x-axis and observing the values on the curves while growing the threshold *R*.

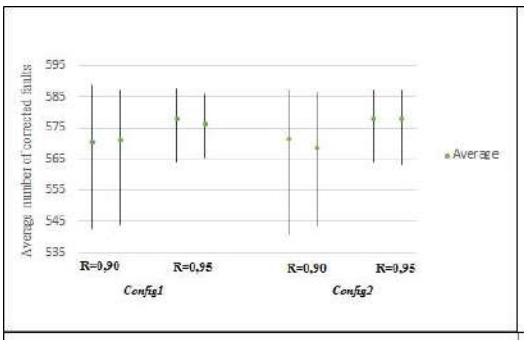
Step 1: Let us assume that all the debuggers may work one hour a day for each of the system functionalities. Then, let us decrease the number of average hours that a debugger may work in order to complicate the search space.

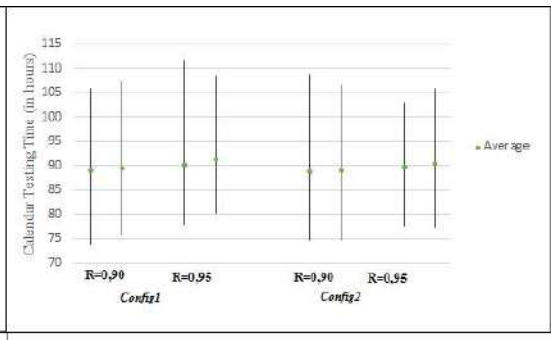
We have generated an additional perturbed configuration by randomly varying the parameters of the nominal values (as done for the Step 0), with the exception of the total amount of testing-effort, *B*, that has randomly decreased of the 10% of the nominal value. We have solved the two optimization models in this new perturbed configuration for a set of values of reliability bound and the average number of hours required for fixing a bug of the functionality *k*, δ_k.

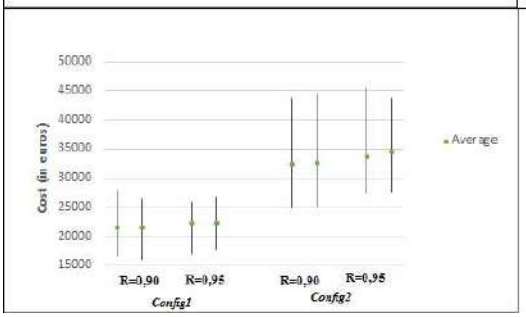
In Figure 14**,** we report the results obtained by the two models with two different values of the average number of hours required for fixing a bug of the functionality k, δ_k. The first configuration corresponds to the one of the nominal instance, whereas the in the second configuration (as shown in Figure 10) we have increased the average number of hours required for fixing a bug of each of the functionalities. More specifically, the figures report the obtained results, where each bar indicates, respectively, the number of corrected faults, the testing time and cost of a model averaged over its new perturbed configuration.

Given a graph represented in Figure 27, each group of two bars - corresponding, respectively, to the base model and our model - refers to the models' results with the perturbed configuration. In particular, each bar - corresponding to the model solution over the configuration with a fixed value of the threshold *R* - reports the

highest, lowest and average number of corrected faults, the testing time and cost obtained.







	Average number of hours required for fixing a bug of the functionality k, δ_k
Config1	$\delta_1 = 2,\ \delta_2 = 2,\ \delta_3 = 2,\ \delta_4 = 2,$ $\delta_5 = 1,\ \delta_6 = 1,\ \delta_7 = 2,\ \delta_8 = 1$
Config2	$\delta_1 = 5,\ \delta_2 = 4,\ \delta_3 = 4,\ \delta_4 = 5,$ $\delta_5 = 5,\ \delta_6 = 5,\ \delta_7 = 5,\ \delta_8 = 5$

FIGURE 27: STEP 1 RESULTS

For each model, the testing cost usually increases in accordance with the reliability required by the system (even thought this increase is more evident for second configuration). Thus, to satisfy the reliability constraint, it is necessary to allocate a greater amount of testing-effort (in man-hours).

The results highlight, in general, that the discrepancies between the two models results starts becoming more evident. In particular, our model starts capture the variation of corrected bugs, the amount of testing time and cost, while modifying the bug assignment activities into the fault correction process of each of the functionalities.

Step 2: Let us assume that all the debuggers may not work one hour a day for each of the system functionalities. We study the sensitivity of the solution to the debugger fixing time values by randomly assigned some of the functionalities to each debugger. By fixing the reliability threshold R to 0.95, for the second configuration of the average number of hours required to fix a bug (see Figure 14), the average number of bugs corrected of corrected faults of our model averaged over its new perturbed configuration (defined in Step 1) decrease from about 578 to about 544.

If we increase the average number of hours required to fix a bug of some of the functionalities (i.e., we set $\delta_1 = 6$, $\delta_2 = 5$, $\delta_3 = 6$, $\delta_4 = 6$, $\delta_5 = 6$, $\delta_6 = 5$, $\delta_7 = 6$, $\delta_8 = 5$), then the average number of bugs corrected of corrected faults of our model averaged over its new perturbed configuration still decreases from about 544 to about 481.

4.5 RELATED WORK

The work related to our research can be divided into four categories: (i) testing resource allocation; (ii) selection of SRGMs; (iv) bug assignment; and (iii) parameters uncertainty.

Testing Resource allocation. In the last years, several research efforts have been devoted to allocate testing resources (e.g., [7], [3]). All these approaches basically provide guidelines to assign appropriate testing resources to a number of relatively small and independent modules (components), which are tested independently during module testing phase. Typically, they express the relationship between reliability and testing resources by using SRGMs. More specifically, these types of reliability models are used for describing the failure occurrence and/or fault removal and consequently aid to enhance the software reliability. Moreover, since failure curves can be either exponential or S-shaped for different modules, flexible SRGMs have also been considered, for example, as done in [7]. In particular, the latter uses a flexible SRGM considering testing effort which, depending upon the values of parameters, can describe either exponential or S-shaped failure pattern of software modules. Testing-effort functions (TEFs) have been introduced (e.g., see [29]) to describe the relationship between the effort expended to test software (e.g., in person-months), and the physical characteristics of the software, such as LOC, etc. In [38], it is shown how to incorporate the logistic TEF [39] into both exponential type, and S-shaped software reliability models. Most SRGMs assume that faults detected during tests will eventually be removed [38].
This assumption, although common in state-of-the-art approaches, might not be realistic. However, a class of related papers deals with this imperfect debugging phenomenon (e.g., see [40], [41], [42]). For example, in [40], general frameworks are proposed for deriving several software reliability growth models based on a non-homogeneous Poisson process (NHPP) in the presence of imperfect debugging and error generation.

Existing approaches for testing resource allocation basically are based on simple optimization models (e.g., in [8] two models are presented that minimize the remaining faults and the amount of testing-effort given the number of remaining faults, respectively) or multi-objective optimization models, for example, maximizing reliability, and minimizing testing cost and testing resource consumed [3]. Different approaches have been adopted such as genetic algorithms in [43], or the gradient projection method and the dynamic programming (a list of these types of works can be found in [3]).

Selection of SRGMs. In the last years the topic of definition, evaluation, and selection of SRGMs has been largely studied (see, e.g., [44] and [45]). Comparative analysis of SRGM models have also been performed in term of goodness of fit, prediction accuracy and correctness, for example, based on failure data sets containing system test failures data, field and open source software defects data [46]. However, although SRGM is probably one of the most successful techniques in the literature, with more than 100 models existing in one form or another, through hundreds of publications [47], in practice, SRGMs encounter major challenges. As remarked in [48], the evaluation of the SRGMs' predictive power in the literature has generally been limited to only the last few

data points (typically last 10% of data) [49] [50]. Moreover, as also claimed in [48], the difficulty of applying SRGMs in industry is compounded with (i) the lack of studies applied to specific industrial domains [49], and (ii) scarce guidelines to select the best SRGMs for a given software process/application. In [48], it has been investigated the application of SRGMs in embedded software domain. In particular, eight established SRGMs have been evaluated on a number of large software projects within the embedded software domain from three different companies.

Bug assignment. The bug assignment problem, related to triage new arriving bug reports to the most qualified developer, has in recent years received increasing attention. An effective bug assignment in large software projects not only requires significant contextual information about both the reported bugs (and the pool of available developers), but also is a time-consuming and tiresome process [13]. Considerable research efforts in the mining software repositories field have concerned bug prediction.

The bug assignment process has been supported by, for example: (i) exploiting the application of information retrieval techniques in order to identify the most appropriate developers [51]; (ii) using expertise models of developers based on previous bug reports [51] [52] or source code contributions [53] ; (iii) applying a machine learning algorithm the open bug repository to learn the kinds of reports each developer resolves [52] ; or (iv) adopting preference elicitation methods to determine the developer's preferences for fixing certain types of bugs [54]. In [13], an auction-based multi-agent mechanism also allows developers to require bugs from the bug triggers; therefore, they can make decisions based on their preferences, expertise, and such.

The problem of resource scheduling for bug fixing can be classified as a special case of the more general resource constrained scheduling problem, which is in general NP-hard [55]. The effectiveness and efficiency of search-based techniques have already been demonstrated for different scheduling related software project management problems (e.g., for project planning in the context of a massive maintenance intervention [56]). However, the application of search techniques for implementing an efficient bug repair policy is very much unexplored [55]. In [55], a genetic algorithm is designed for scheduling developers and testers to bug-fixing tasks considering both human properties (skill set, skill level and availability) and bug characteristics (severity and priority). Also, industrial software defect prioritization techniques, in general, suffer of lack of multi-optimization techniques [57].

Another class of related papers deals with automated debugging techniques that aim to help developers locate and understand the cause of a failure (e.g., [58]). In particular, statistical-fault-localization techniques have been extensively investigated (see [58] for an overview on these types of techniques and other ones like anomaly detection, and experimental debugging). Other papers are focused on assisting developers in changing programs to fix bugs. For example, in [59], based on a machine learning technique, a tool has been designed for computing and reporting a prioritized list of bug-fix suggestions for a given debugging situation at a program statement that is suspected of being faulty.

Parameters uncertainty. Other challenges related to the testing allocation are represented by parameters uncertainty. In fact, software testing activity is fraught with a not negligible uncertainty relates to values of parameters such as operational profile, the expected number of initial faults, the fault detection rate per unit testing-effort (SRGMs input), fault fixing time, etc. Several research efforts have also been spent in order to deal with parameters' uncertainty in software quality domain (e.g., in component reliability estimates [60], or in the performance modeling and analysis process [9]) adopting, for example, a robust optimization approach [10], or a bayesian approach [61]. Moreover, for example, fuzzy mathematical methods have been used to represent the uncertainty parameters (e.g., as done in [62]) of an alternative architecture. The fuzzy paradigm has also been used in [63], wherein it is addressed uncertainty involved in estimated parameters of SRGM in imperfect debugging environment. Therefore, although there is a growing interest in handling uncertainty, in practice, uncertainty of all the parameters of a software testing activity is not typically addressed.

With respect to the state-of-art, the following major aspects characterize the novelty of the approach:

- This is the first work (to the best of our knowledge) that enables practitioners to maximize the effectiveness of the testing activity using an optimization framework, which allows dynamically to model: (i) fault detection and correction processes of systems functionalities (modules) through the SRGMs that best fit the actual testing data, (ii) testing cost/time constraints, and (iii) parameter-specific uncertainties phenomenons. So that the systems functionalities (modules) with shorter time (budget) are tested and that reveled bugs are fixed earlier.
- We have explicitly considered the bug assignment activity in the fault correction process (typically not done in the existing works). In particular, this work has showed that (for a large software system) the bug assignment may be a key factor for a trustworthy prediction of the fault correction process of the single functionalities (software modules), as well as of the reliability of the whole system.
- The proposed approach does not rely on a specific development process or testing practice (e.g., in testing unit).
- We have provided guidelines for practitioners. We have provided support for their testing allocation decisions based on cost, time, and software quality. In particular, our approach helps to: (i) select (and use) SRGMs in order to make the software testing process more effective; and (ii) handle parameters uncertainty, which, as shown through our real world software project, plays a critical role in accurately describing testing resource allocation process. More specifically, we have shown that the handling of uncertainty is a key factor for a trustworthy prediction of the reliability of a software system, and leads an optimization model to a more precise (and less pessimistic) estimation of the system reliability, as well as to a more effective and efficient testing resource allocation activity. It is well known that SRGMs sometimes show good performance in terms of predictability

of the software reliability, but sometimes they do not. This fact may be, in particular, caused by insufficient information on how the software has been developed, maintained, and operated [64].

- We have instantiated the optimization model for the fault correction with the bug assignment activity prediction, but its elements (e.g., cost function and reliability constraints) combined with the method for uncertainty analysis could be re-used in another phase of the testing process. This adoption may require specializing (appropriately modifying) the model in order to capture typical aspects of the new phase. Testing-effort allocation prediction under testing-effort time/cost and reliability constraints with uncertain model parameters, for example, could be used for enhancing existing approaches (such as that one in [55]) for scheduling developers/testers to activities to be performed to fix a bug repository.

5 OPTIMAL REGRESSION FUNCTIONAL TESTING

Regression testing is the process of validating modified software to provide confidence that (i) the changed parts of the software behave as intended, and (ii) the unchanged parts have not been adversely affected by the modifications [65].

Research in regression testing has seen a flourish in the past years, in particular in the fields of new approaches, tools, and techniques to reduce the cost of reusing the test suite that was used to test the original version of the software. A quite extensive list of these approaches can be found in [66] and [67]. However, the key tasks of testing cost reduction methods are commonly: (i) regression test selection - selecting subset of existing test cases to run on the modified software (e.g., [68], [69], [70], and [71]); (ii) regression test suite minimization - reducing the test suite size to a minimal subset to maintain the same level of coverage as the original test suite; and (iii) regression test suite prioritization - finding an ideal order of test cases according to some criteria, such that test cases with higher priority are executed earlier than ones with lower priority [72].

Although used extensively in industry, regression testing is challenging from both a process management as well as a resource management perspective. In fact, putting the proposed techniques into practice has been a challenge [72].

In Section 5.1, we introduces related work. In Section 5.2, we present an overview of our approach.

5.1 RELATED WORK

In the last years the topic of software testing has been studied in several communities and from different perspectives (see, e.g., [73] for a look into architecture-based testing techniques, or the survey in [74] of methodologies for automated software test case generation).

In particular, a lot of research efforts has been spent for regression testing (e.g. see survey [66]). In this work, we focus on regression test suite prioritization, which is highly relevant in general to industry (and in particular, for our industrial partner). Therefore, hereafter, we review works appearing in the literature dealing with regression testing prioritization.

Several techniques have been introduced for using test execution information to prioritize test cases. In [75], a comparison of such techniques, aimed to evaluate their ability to improve rate of fault detection, has been performed by conducting several empirical studies. More specifically, three categories of techniques have been considered, i.e., techniques ordering test cases based on their (i) total coverage of code components, (ii) coverage of code components not previously covered, and (iii) estimated ability to reveal faults in the code components that they cover. Several new controlled experiments and case studies have been

performed in [76]. In particular, building on results presented in [75] and focusing on the goal of improving rate of fault detection, the authors in [76] have addressed additional questions (e.g., related to the techniques' effectiveness when targeted at specific modified versions, or the trade-off between the fine granularity and coarse granularity prioritization techniques).

Research effort has been also devoted for defining metrics to quantify and compare the rates of fault detection of test suites [77], [78]. In [79], a more general metric has been defined for measuring rate of fault detection that accounts for varying test case and fault costs.

Another class of related papers deals with prioritization techniques that are driven by requirements with higher priority, or operate in the presence of time constraints (e.g., [80], [81], [82] discussed below).

In [80], a regression testing approach is proposed, where test cases are prioritized such that the test cases for requirements with higher priority are executed earlier during system test. In particular, four factors (i.e., requirements volatility, customer priority, implementation complexity, and fault proneness) are used to analyze and assign value to each requirement.

The work in [81] presents initial results of an empirical study on using historical test execution data to prioritize test case selection in a constrained regression testing process. In particular, the work evaluates how several RTS techniques perform under severe time and resource constraints.

In [82], it is presented a regression test prioritization technique that uses a genetic algorithm to reorder test suites in light of testing time constraints.

The genetic algorithms (to determine the most effective order) have also been leveraged in [83]. Specifically, this work proposes a method of cost-cognizant test case prioritization based on the use of historical records, which are gathered from the latest regression testing.

A comparison of search algorithms for regression test case prioritization, based on code coverage (including block coverage, decision (branch) coverage, and statement coverage) has also been performed in [84]. More specifically, the work presents results from an empirical study of the application of several greedy, metaheuristic, and evolutionary search algorithms to six programs, ranging from 374 to 11,148 lines of code for three choices of fitness metric.

Several coverage-based test case prioritization techniques have been developed, which typically use either a total strategy or an additional strategy. In [85], it is proposed a unified test case prioritization approach that encompasses both the total and additional strategies. The work has also proposed extensions to enable the use of differentiated probabilities that test cases can detect faults for methods and the use of static coverage information as well as dynamic.

There was previous work, which has exploited the combination of code coverage analysis and the change impact analysis. For example, in [86], a procedure level

coverage regression test based on change-based test selections method has been experimentally applied to an open source web browser engine project WebKitit. Moreover, the work also experimented test case prioritization strategies (based on changes) to reduce the testing time when the selection is too large.

Approaches for particular types of applications (such as for software product lines [87]) or testing strategies (e.g., model-based testing [88]) have also been introduced, as well as the use of methods (e.g., information retrieval ones [89]) have been exploited, for example, in order to address coverage profiling overhead (in terms of time and space) and potential problems associated with the imprecisions of static program analysis. Research effort has also been done for improving regression testing in continuous integration development environments [90]. In particular, the work in [90] has introduced two regression testing techniques (for testing selection and prioritization, respectively) that use readily available test suite execution history data to determine what tests are worth executing and executing with higher priority.

5.2 OVERVIEW OF OUR APPROACH

A representation of the high-level workflow of the proposed approach is presented in Figure 28.

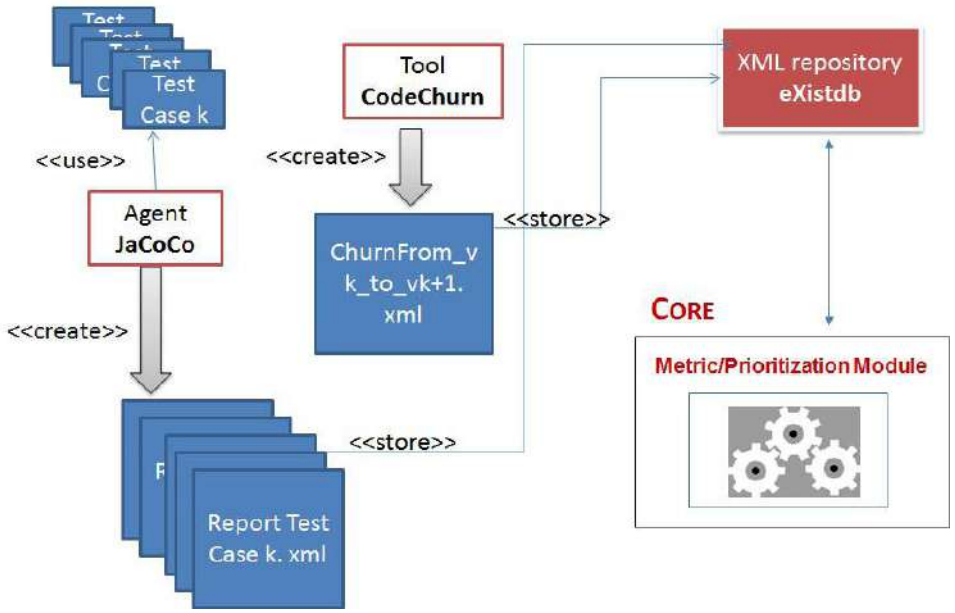


FIGURE 28: *HIGH-LEVEL APPROACH OVERVIEW*

Our approach is mainly based on the analysis of code coverage and code churn, which is collected for each of the version of a software product. Such information is stored in a database, our implementation makes use of eXist-db database, which

is an open source NoSQL database and application platform built on XML technology. [10]

Coverage information is collected by a JaCoCo agent, an open source toolkit for measuring and reporting Java code coverage.[11] A JaCoCo report is a xml document having the structure depicted in Figure 29.

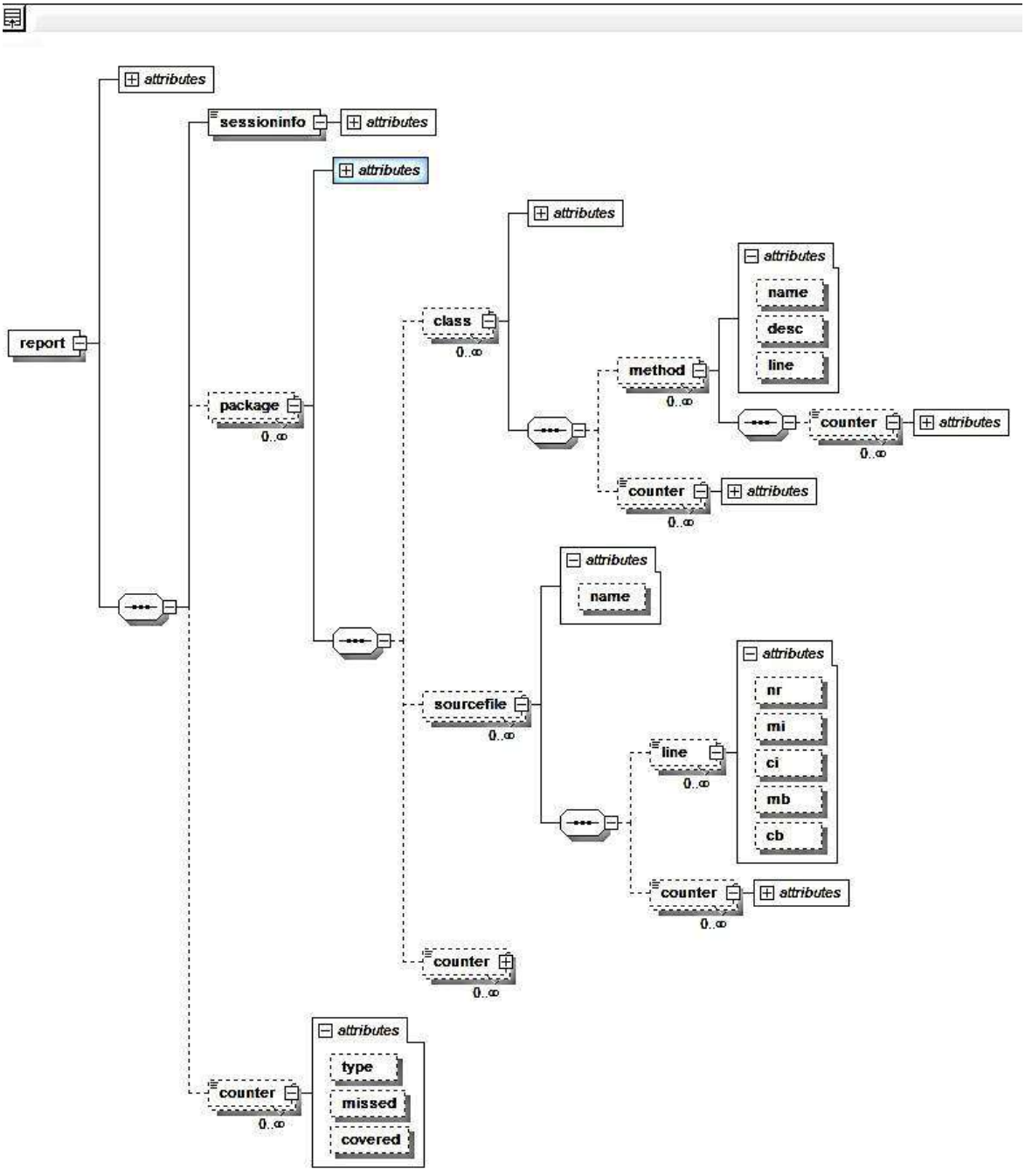


FIGURE 29: OUTPUT OF THE JACOCO TOOL: CODE COVERAGE ANALYSIS

[10] The eXist-db database can be obtained freely from [91].

[11] The JaCoCo tool can be obtained freely from [92].

Churn metrics are collected by CodeChurn Tool. It is a proprietary tool of ASSIOMA.net [93], which exploits the Sonar tool [94] for metrics evaluation. More details on the churn code metrics and the Sonar tool can be found in Deliverable D2.2 [95]. A CodeChurn report is a xml document having the structure depicted in Figure 30.

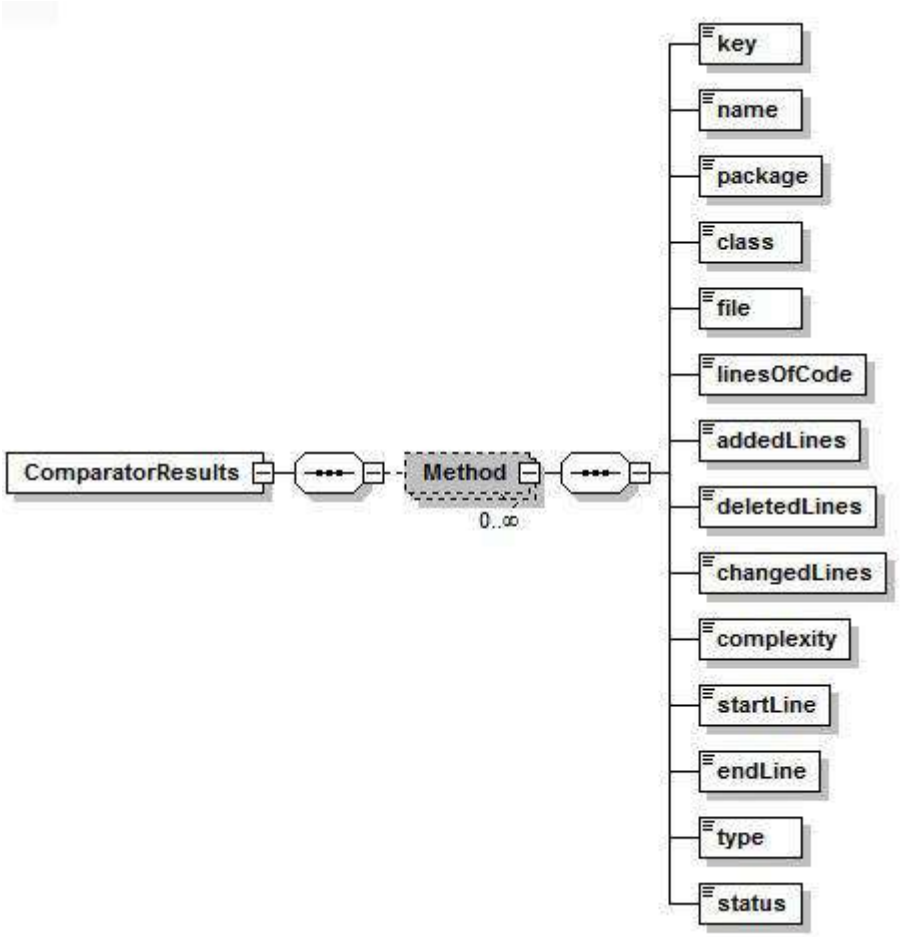


FIGURE 30: OUTPUT OF THE CODE CHURN TOOL: CODE CHURN ANALYSIS

In Section 5.2.1, we provide more details on the JaCoCo tool's output, whereas in Section 5.2.2, we describe in more detail the Code Churn tool's output. Finally, in Section 5.2.3, we give an overview of the Metric/Prioritization module, which represents the core of our approach.

5.2.1 The JaCoCo Tool

The JaCoCo tool provides code coverage analysis in Java VM based environments. It is based on Bytecode instrumentation; therefore it is very helpful in situations where the source code is not available.

As illustrated in Figure 29, the JaCoCo tool allows to collect coverage analysis at different level of granularity, resulting in the following coverage measures.

- *Instructions*, namely single Java byte code instructions. In particular, instruction coverage is related to the amount of code that has been executed or missed.
- *Branches* for all *if* and *switch* statements. In particular, the total number of such branches in a method are counted so as to determine the number of executed or missed branches.
- The *Cyclomatic Complexity* is estimated for each non-abstract method, classes, packages, and groups.
- *Lines*. Coverage information for individual lines are calculated for the class files that have been compiled with debug information. In particular, if at least one instruction that is assigned to a certain source line has been executed, then the source line is considered executed.
- *Methods*. A non-abstract method contains at least one instruction, and is considered as executed when at least one instruction has been executed. Notice that constructors and static initializers are also counted as methods, because JaCoCo is based on Bytecode instrumentation.
- *Classes*. If at least one the methods of a certain class has been executed, then the class is considered as executed.

More details on the tool can be found in [92]. The supported reports formats are HTML, XML, and CSV. In our implementation, we have chosen the XML format.

An extensive list of code coverage tools for java can be found in [96].

5.2.2 The Code Churn Tool

As illustrated in Figure 30, the Code Churn tool allows collecting churn code analysis at different level of granularity. In particular, the tool evaluates the (i) Total added, modified and deleted LOC, and (ii) Cyclomatic complexity.

5.2.3 Metric/Prioritization Module

A primary input to the Metric/Prioritization Module is represented by an XML-based structure collecting churn metrics and coverage metrics. In fact, for each of the test case, churn metrics and coverage metrics are joined in a common structure depicted in Figure 31. More specifically, the XML output files of the JaCoCo tool, related to the code coverage analysis, and Code Churn tool, are merged. To this extend, we have exploited the proprietary RaptorXML tool, which is a hyper-fast

XML and XBRL processor. A 30-day trial version of RaptorXML can be downloaded from [97].

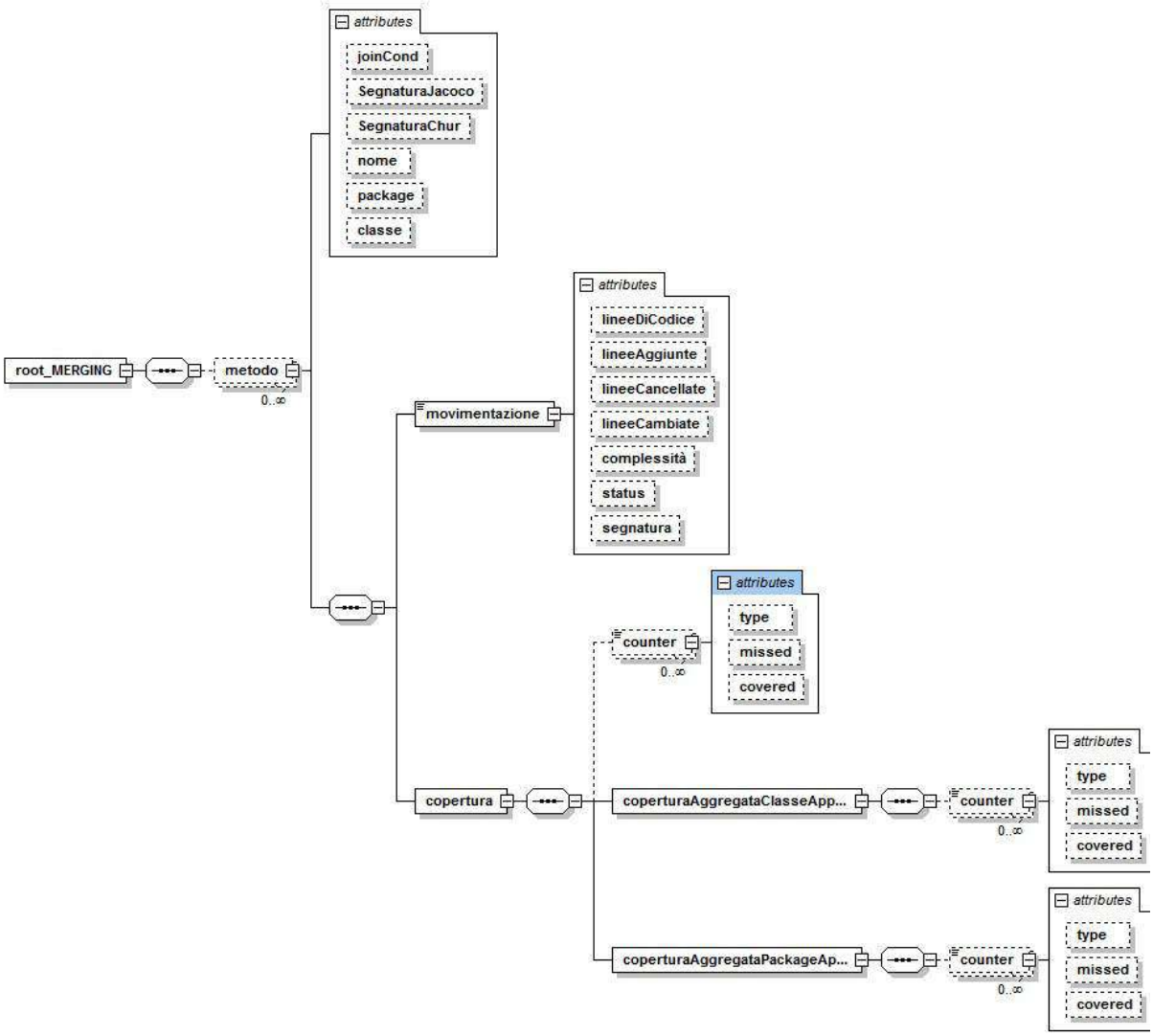


FIGURE 31: OUTPUT OF THE RAPTORXML TOOL: MERGING OF THE COVERAGE AND CHURN ANALYSIS

The data models of the JaCoCo tool (see Figure 29) and the Code Churn Tool (see Figure 30) are precisely the ones used in eXist-db database.

Table 16 and Table 17 summarize, respectively, the input and the output of the Metric/Prioritization module. Specifically, the Metric/Prioritization Module processes the XML-based structures, and assigns priority to the test cases. Priority assignment involves applying a function that seeks to capture the relationship among the test cases, the code coverage, and the churn analysis. The goal of the prioritization we are interesting in is that of considering in order of relevance (a) tests case potentially covering changed parts of the product (b) test cases which guarantee the best coverage. Specifically, we have introduced two parametric algorithms inspired on standard Total statement coverage prioritization and the

Additional variant. Details on the implementation of these algorithms can be found in the next section.

TABLE 16: INPUT OF THE METRIC/PRIORITIZATION MODULE

Source	Data Type	Description
User	Source code of system versions (java code)	Our approach is mainly based on the analysis of code coverage and code churn, which is collected for each of the version of a software product. Such information is stored in a database, our implementation makes use of eXist-db database, which is an open source NoSQL database and application platform built on XML technology.
User	Test cases	Test cases to prioritize
JaCoCo tool	Coverage information	Analysis of code coverage is collected for each of the version of a software product. Specifically, coverage information is collected by a JaCoCo agent, an open source toolkit for measuring and reporting Java code coverage.
CodeChurn Tool	Code churn analysis	Churn metrics are collected by CodeChurn Tool. It is a proprietary tool of ASSIOMA.net, which exploits the Sonar tool for metrics evaluation

TABLE 17: OUTPUT OF THE METRIC PRIORITIZATION MODULE

Decision	Description
Test cases prioritization	A primary input to the Metric/Prioritization Module is represented by an XML-based structure collecting churn metrics and coverage metrics (obtained, respectively, with JaCoCo and CodeChurn tool). In fact, for each of the test case, churn metrics and coverage metrics are joined in a common structure. More specifically, the XML output files of the JaCoCo tool, related to the code coverage analysis, and Code Churn tool, are merged. To this extend, we have exploited the proprietary RaptorXML tool, which is a hyper-fast XML and XBRL

	processor. The Metric/Prioritization Module processes the XML-based structures, and assigns priority to the test cases. Priority assignment involves applying a function that seeks to capture the relationship among the test cases, the code coverage, and the churn analysis. The goal of the prioritization we are interesting in is that of considering in order of relevance (a) tests case potentially covering changed parts of the product (b) test cases which guarantee the best coverage.

5.2.4 Test prioritization

5.2.4.1 Churn Coverage Prediction Prioritization

In this section we consider predictive prioritization techniques which exploit both coverage and churn information. In this case we do not consider bursts but focus our attention only on the two last versions V_{m-1} and V_m of a sequence of versions $\langle V_1, \dots, V_m \rangle$. In this case we assume that tests have been already executed on V_{m-1} (i.e. coverage metrics are available on that version) but have not yet executed on version V_m for which only churn data are available.

The challenge of predictive prioritization is that of estimating a good prioritization of test cases for version V_m by exploiting churn data and coverage data collected for version V_{m-1}.

The goal of the prioritization we are interesting in is that of considering in order of relevance (a) tests case potentially covering changed parts of the product (b) test cases which guarantee the best coverage. To this purpose we propose suitable adaptations of two well-known prioritization techniques, namely the Total Statement coverage prioritization and the Additional Statement Coverage Prioritization.

Actually, we introduce two parametric algorithms inspired on standard Total statement coverage prioritization and the Additional variant. This algorithms exploit structured coverage information for test cases referred in the following as coverage increment. Intuitively, the coverage increment of a test case depends on the current state of coverage and gives the contribution of coverage split into three components: the contribution for changed parts, for deleted parts and for unchanged parts. By introducing suitable ordering criteria for coverage increments we are able to define variants of the prioritization algorithm.

Let $M(V)$ be a coverage report for the version V of a product, namely a .xml structure recording the coverage information after the execution of a (possibly empty) set of test case. With $M_0(V)$ we denote the initial coverage report corresponding to the execution of an empty set of test cases. For a test T, a version V and a coverage report $M(V)$ let be $Inc(T,M)$ be the quadruple $\langle C, D, U, T \rangle$ where

- C is the number of instructions of methods that will change w.r. to version V coveredby the execution of T and uncovered in $M(V)$;
- D is the number of instructions of methods that will be deleted w.r. to version V covered by the execution of T and uncovered in $M(V)$;
- U is the number of instructions of methods that will remain unchanged w.r. to version V covered by the execution of T and uncovered in $M(V)$.

$Inc(T, M)$ **gives** the coverage increment with respect to the coverage report M after the execution of the test case T. Such a tuple is called coverage increment tuple. Notice that is $Inc(T, M_0)$ precisely the tuple $\langle \vec{C}(V,T), \vec{D}(V,T), \vec{U}(V,T), T \rangle$. For a set of test cases Z, $MaxInc_{\preccurlyeq}(Z, M)$ gives the test case in Z which guarantees the greatest coverage increment among all the test cases in Z, namely $MaxInc(Z, M)$ is the test case $T \in Z$such that $Inc(T, M) = max_{\bar{T} \in Z}\{Inc(\bar{T}, M)\}$ where *max* is computed with respect to the parametric ordering of quadruples $\preccurlyeq$

Let us consider now the Churn Total Statement Coverage Prioritization. The pseudocode is reported in Figure 32.

Parameters:
Set of test cases $\mathcal{TC}$
Consecutive versions $\langle V_1, \ldots, V_m \rangle$
Version to test V_m
Ordering $\preceq$
Variables:
A sequence of increment tuples Seq
0. $Seq = \epsilon$
1. **for** $T \in TC$ to **do**
2. $InsertOrdedered_{\preceq}(Seq, Inc(T, \mathcal{M}_0(V_{m-1}))$
3. **return** $Test(Seq), p)$

FIGURE 32: ALGORITHM 1: CHURN TOTAL STATEMENT COVERAGE PRIORITIZATION

The Churn Total Coverage Prioritization can be easily obtained by ordering under the parametric ordering $\preccurlyeq$ the coverage increment tuples of all the considered test cases (the function Test applied to a sequence of coverage increment tuples simply gives the sequence of projection of the test name component of each tuple).

The standard Total instruction coverage prioritization (which do not consider churn information) can be defined by considering the ordering $\prec$ defined as follows

$$\langle C, D, U \rangle \prec_{St} \langle C', D', U' \rangle \text{ iff } C + D + U \leq C' + D' + U'$$

In [98] some prioritization criteria sensitive to churn are introduced. For instance, the General strategy is intended to cover most procedures besides the changed ones under the assumption that test cases with higher overall coverage are better. The opposite of General is the Specific strategy which is intended to cover least procedures besides the changed ones. The specific strategy selects those test cases first which cover little outside of the changes. In our setting we can define analogous strategies working at the granularity level of instructions instead of granularity level of methods. For instance, the principles of the general strategy can be enforced by the following ordering $\preccurlyeq_{Gen}$ defined as follows

$$\langle C, D, U\rangle \prec_{Gen} \langle C', D', U'\rangle \text{ iff } \frac{U}{D+C+U} \leq \frac{U'}{D'+C'+U'}.$$

On the opposite, the principles of the specific strategy can be enforced by the following ordering $\prec_{Spec}$ defined as follows

$$\langle C, D, U\rangle \prec_{Spec} \langle C', D', U'\rangle \text{ iff } \frac{D+C}{D+C+U} \leq \frac{D'+C'}{D'+C'+U'}.$$

Finally, we consider a kind of ordering $\preccurlyeq_{lex}$ which prioritize first the coverage of changed parts if relevant and than that of coverage of unchanged part if the coverage increment of changed parts can be considered equivalent (a kind of 'lexicographic order between coverage of changed parts and coverage of unchanged parts). The definition of the ordering $\preccurlyeq_{lex}$ depends on a parameter $\alpha \geq 0$ which used to determine when the amount of coverage can be considered equivalent.

The ordering $\prec_{Lex}$ defined as follows

$$\langle C, D, U\rangle \prec_{Lex} \langle C', D', U'\rangle \text{ iff } \begin{cases} \text{either } \mid (C+D) - (C'+D') \mid < \alpha \text{ and } U < U' \\ \text{or } \quad (C'+D') - (C+D) \geq \alpha \end{cases}$$

Let us consider now the Churn Additional Coverage Prioritization. The pseudocode is reported in Figure 33.

```
Parameters:
Set of test cases TC
Consecutive versions ⟨V_1, ..., V_m⟩
Version to test V_m
Ordering ≺
Variables:
A sequence of test names Seq
A set of test names Z
A coverage report M
0.  Seq := ϵ; Z := TC; M := M_0(V_{m-1});
1.  While Z ≠ ∅ do
    Loop
2.    T := MaxInc_≺(Z, M));
3.    Append(Seq, T);
4.    M := AddCover(T, M)
5.    Z := Z \ {T}
    End Loop
6.  return Seq, p)
```

FIGURE 33: ALGORITHM 2: CHURN ADDITIONAL STATEMENT COVERAGE PRIORITIZATION.

The function $AddCover(T, M)$ gives as a result a coverage report obtained by adding to *M* the coverage information of test case *T*.

5.2.4.2 Backward Churn Prioritization

In the previous section we have considered algorithms for predictive (forward) prioritization which has to be considered in absence of coverage information for changed parts. If the test cases have been executed at least once in the last version V_m the prioritization can be recomputed taking into account also coverage information. In this case we can consider the same strategies seen for predictive prioritization with a slight modification of the concept of coverage increment tuple called *backward coverage increment tuple.*

For a test *T,* a version *V* and a coverage report *M(V)* let be $IncB(T, M)$ be the quadruple $\langle C, A, U, T\rangle$ where

- *C* is the number of instructions of methods changed in *V* w.r. to the previous version covered by the execution of *T* and uncovered in *M(V);*
- *A* is the number of instructions of methods added in *V* w.r. to the previous version covered by the execution of *T* and uncovered in *M(V);*
- *U* is the number of instructions of methods unchanged in *V* w.r. to the previous version covered by the execution of *T* and uncovered in *M(V).*

$IncB(T, M)$ gives the coverage increment with respect to the coverage report *M* after the execution of the test case *T*.

Notice that the backward coverage increment tuple simply replaces the coverage of methods which will be deleted with the coverage of methods which are added. In this case $IncB(T, M_0)$ is precisely the tuple$\langle \overleftarrow{C}(V,T), \overleftarrow{A}(V,T), \overleftarrow{U}(V,T), T\rangle$. For backword increment coverage tuple, the analogous of $\prec_{St}, \prec_{Gen}, \prec_{Spec}$ and $\prec_{Lex}$, written $\prec_{St\,B}, \prec_{GenB}, \prec_{SpecB}$ and $\prec_{LexB}$, respectively, by simply replacing in the definitions the metrics deleted methods with the metrics of added methods.

Therefore, a predictive (forward) prioritization can be used to suggest the first regression test for a new version. A backward prioritization can be used for the next stages (after the first). The backward prioritization allows in addition to check the predictive power of forward prioritization. The idea is that a good prediction should be very similar to the ordering of test output by a backward prioritization.

To measure the distance of two prioritizations (two orderings of the same set of test cases) we shall consider, for instance, the following definition. A prioritization of a test suite *TC* is an bijective function $pr: TC \rightarrow \{1, \ldots, |TC|\}$ (intuitively *pr(T)* gives the position of the test $T \in TC$ in the prioritization). Given two prioritizations pr_1 and pr_2 for *TC* with = n, the distance of two prioritizations is given by

$$D(pr_1, pr_2) = \frac{2}{|TC|^2} \sum_{T \in TC} |pr_1(T) - pr_2(T)|$$

Notice that the distance of two equal prioritization is 0. The constant $\frac{2}{|TC|^2}$ gives an upper bound for the greatest possible distance.

5.2.5 Experimental Results

For the experiments we have considered SIR [99], a repository of software-related artifacts meant to support rigorous controlled experimentation with program analysis and software testing techniques, and education in controlled experimentation. For the experimentation we have considered Java products having a meaningful number of lines of code, of versions and cardinality of test suit. The chosen products are SIENA and ANT whose attributes are depicted in Table 18. Siena (Scalable Internet Event Notification Architecture) is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks, responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering those notifications to the clients via access points [100]. The associated test suite guarantees a complete method coverage (not a complete statement coverage). Ant is a Java-based build tool supplied by the open source.

TABLE 18: CASE STUDIES

	Versions	Test cases	Classes	LoC
Siena	8	567	26	6035
Ant	11	28	627	80500

In Table 19 and Table 20, we report the churn metrics provided by the tool ChurnTool (we consider eight versions for both Siena and Ant).

TABLE 19: CHURN METRICS FOR ANT

Versions	Methods Prev	Methods Next	Changed Meth.	Deleted Meth.	Added Meth.	Total Meth.	Total Churn	Percent. Churn
V_0 to V_1	1655	2538	322	147	1030	2538	58,6	1,26
V_1 to V_2	?	?	?	?	?	?	?	?
V_2 to V_3	3888	3897	57	2	11	3897	1,96	0,02
V_3 to V_4	?	?	?	?	?	?	?	?
V_4 to V_5	5731	5851	303	45	165	5851	7,23	0,1
V_5 to V_6	5851	5877	17	2	28	5877	1	0,01
V_6 to V_7	?	?	?	?	?	?	?	?
V_7 to V_8	7637	7617	54	24	4	7617	0,73	0,01

TABLE 20: CHURN METRICS FOR SIENA

Versions	Methods Prev	Methods Next	Changed Meth.	Deleted Meth.	Added Meth.	Total Meth.	Total Churn	Percent. Churn
V_0 to V_1	196	187	2	9	0	187	2,58	0,05
V_1 to V_2	187	243	1	0	56	243	24,45	0,31
V_2 to V_3	187	197	3	0	10	197	2,53	0,07
V_3 to V_4	197	198	1	0	1	198	0,72	0,01
V_4 to V_5	198	198	3	0	0	198	0,48	0,02
V_5 to V_6	198	198	1	1	1	198	0,93	0,02
V_6 to V_7	198	194	9	4	0	194	1,61	0,07

Analysis of the results

In order to show the effectiveness of the combination of coverage and churn information we here illustrate the results that we have obtained from the prioritization which optimizes either coverage or coverage of changed parts in the next software version.

For the Ant and Siena systems, we have prioritized the test cases by using coverage and churn information. More specifically, for a version V_k, we have prioritized the test cases by using the relationship among the test cases, the code coverage, and the churn analysis.

The experiments were run on a Ubuntu Linux 12.04 workstation equipped with a Intel Core i7 (2 MB of cache memory and 8 GB RAM DDR3).

InFigure 34, we report the obtained results for the Siena system. For each version, we have prioritized the test cases, and estimated the coverage of changed parts of the first 50, 150, 250, and 350 test cases of its prioritized test suite. Each bar indicates the number of instructions (i.e., single Java byte code instructions) of changed parts covered by the test cases. Therefore, we have measured the predictive power of test cases as a function of the changed parts. To sake of comparison, we have also estimated the coverage of the whole test suite.

FIGURE 34: COVERAGE OF TEST CASE PRIORITIZATION FOR THE SIENA SYSTEM

In Table 21, we report the detailed results. Each cell reports the resulting number of instructions of changed parts covered by the test cases for a certain version (row) and a certain number of test cases (column).

TABLE 21: COVERAGE FOR SIENA

Versions Versions	Coverage whole test suite	Coverage first 50 tests	Coverage first 150 tests	Coverage first 250 tests	Coverage first 350 tests
V_0 to V_1	3446	839	985	1385	1392
V_1 to V_2	3460	1934	2576	2935	2945
V_2 to V_3	3291	1342	2705	2709	2709
V_3 to V_4	5319	2729	4777	4777	5296
V_4 to V_5	5346	2798	2868	3039	3107
V_5 to V_6	5211	3426	4799	4826	5012
V_6 to V_7	3347	2449	3142	3244	3314

Similarly, in Figure 35, we report the obtained results for the Ant system. For some versions, we have prioritized the test cases, and estimated the coverage of changed parts of the first 8, 10, 14, 18, and 24 test cases of its prioritized test suite. Each bar indicates the number of instructions (i.e., single Java byte code instructions) of changed parts covered by the test cases. To sake of comparison, we have also estimated the coverage of the whole test suite.

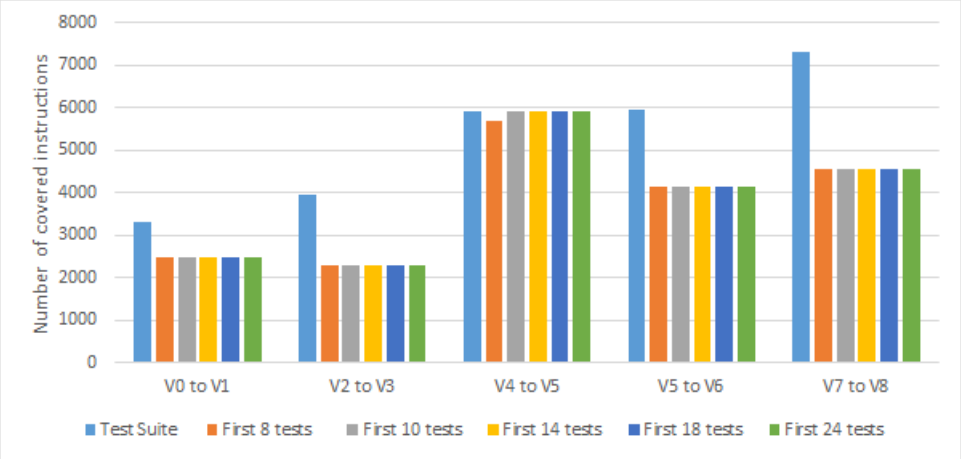


FIGURE 35: COVERAGE OF TEST CASE PRIORITIZATION FOR THE ANT SYSTEM

In Table 22, we report the detailed results. Each cell reports the resulting number of instructions of changed parts covered by the test cases for a certain version (row) and a certain number of test cases (column).

TABLE 22: COVERAGE FOR ANT

Versions Versions	Coverage whole test suite	Coverage first 8 tests	Coverage first 10 tests	Coverage first 14 tests	Coverage first 18 tests	Coverage first 24 tests
V_0 to V_1	3308	2458	2458	2458	2458	2458
V_2 to V_3	3936	2272	2272	2272	2272	2272
V_4 to V_5	5928	5676	5928	5928	5928	5928
V_5 to V_6	5934	4153	4153	4153	4153	4153
V_7 to V_8	7295	4537	4537	4537	4537	4537

The results highlight, in general, that the predictive power of test cases (as a function of the changed parts) almost always increases while increasing the number of selected test cases. For example, for the Siena system (see Figure 34), the number of instructions of changed parts covered by the test cases (except in two cases) almost always increases while increasing the number of test cases. Moreover, the discrepancies among test cases (i.e., their predictive power) become more evident as the number of changes increases (e.g., for versions with higher values of code churn metrics), such as for the version V_2 of the Siena system (see Table 21). On the other hand, the predictive power of test cases do not show discrepancies in case of small increase in the number of test cases or changes. For example, for a given version of the Ant system, the predictive power of test cases does not essentially change.

6 ARCHITECTURAL DECISION-MAKING

The prediction of the software architecture 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 approaches can be introduced in each phase. A major issue in this direction is that Quality of Service (QoS) attribute cannot be analyzed separately, because they (sometime adversely) affect each other. Therefore, approaches aimed at the tradeoff analysis of different attributes have been recently introduced (e.g., reliability vs cost, security vs performance).

Our work has been focused on ***modeling and analysis of QoS tradeoffs of a software architecture based on optimization models***. A particular emphasis has been given to two aspects of this problem: (i) the mathematical foundations of QoS tradeoffs and their dependencies on the static and dynamic aspects of a software architecture, and (ii) the automation of architectural decisions driven by optimization models for QoS tradeoffs. Our major contribution is to show how effectively optimization modeling techniques can capture relevant aspects of the architectural decision-making process in different lifecycle phases, thus representing a very relevant support for the software engineers tasks. We have also given a tutorial on this topic [101].

In the book chapter [102], in the context of a waterfall development process, we implement three models: one for the architectural design (i.e. the software architecture driven model applicable before the release of a system), one for the implementation/deployment phase (we show how the QoS of a software architecture depends on the hardware architecture), and one for the maintenance phase (i.e. the software architecture driven model applicable after the release of a system). In order to show the usefulness of our approach, we run these models on an example coming from the domain of medical information systems.

In this chapter, we have also presented a general optimization model that minimizes the total costs subject to constraints on the level quality of the software architecture. The model can be adopted in (specialized for) one of the lifecycle phases by leveraging available information and parameters, the level of detail of which obviously increases as the development progresses. Then, each specialized form of the general model can be either separately used and solved, if required in a certain lifecycle phase, or used in pipeline feeding with each other, as we will show in our example. In Section 6.1, we report the formulation of this general optimization model by discussing typical architectural decisions, which can be supported by using optimization models.

Our work has also been focused on the ***automation of the support for the decisions that software architects make after deployment***. This approach

is based on an optimization model whose solution suggests the "best" actions to be taken according to a given change scenario (i.e., a set of new requirements that induce changes in the structural and behavioral aspects of the software architecture).

In particular, in [103], we introduce a framework named SHEPhERd (Software arcHitecture Evolution based on cost, PErformance and Reliability), which is composed of a UML case tool, a model builder and a model solver.

SHEPhERd is based on an optimization model that suggests the "best" actions to be taken upon a certain change scenario arising. A change scenario is a set of new requirements that induce changes in the structural and behavioral aspects of the software architecture. In particular, in our model, for each new requirement in a change scenario we consider different sets of evolution actions (called evolution plans) that are able to guarantee these new requirements. We aim to obtain a set of decisions that lead to the definition of a new architecture that minimizes cost, while keeping the reliability and the response time within certain thresholds. In Section 6.2, we describe the main features of the SHEPhERd framework.

In Section 6.3, we introduce the SAQO (System Adaptation with Quality Optimization) framework, which extend the SHEPhERd framework.

6.1 A GENERAL FORMULATION FOR ARCHITECTURAL DECISIONS VS QUALITY CONSTRAINTS

In this section, we report the general optimization model presented in [102].

The model minimizes the total costs subject to constraints on the level quality of the software architecture.

Let $S = \{u_1, \cdots, u_n\}$ be a software architecture made of n software units u_i $(1 \leq i \leq n)$ the composition of which results in services that the system offers to users.

Since the proposed model may support different lifecycle phases, we adopt a general definition of software unit: it is a self-contained deployable software module containing data and operations, which provides/requires services to/from other elementary elements. A unit instance is a specific implementation of a unit. For each unit u_i, let J_i be the set of instances available by vendors and $\overline{J_i}$ the set of possible options for developing the instance in-house. Let u_{ij} be the j-th instance of $J_i \cup \overline{J_i}$.

Architectural Decisions. The analysis of the QoS tradeoffs is a broad decision-making process that consists of a set of actions aiming to modify the static and dynamic structure of the software architecture. The decisions within the different life-cycle phases are basically related to the following software actions:

1. ***Introducing new software units***: One or more new software units may be embedded

into the system.[12] We call $NewS$ the set of new available software units that can provide different functionalities.

2. ***Replacing existing unit instances with functionally equivalent ones available on the market*:** The employed instance u_{ik} of a software unit u_i may be replaced with an element of the set J_i, i.e., with of the instances available for it on the market (e.g. a Commercial-Off-The-Shelf (COTS) component/web service).We assume that all the instances in J_i are functionally compliant with u_{ik}, i.e., each of them provides at least all services provided by u_{ik} and requires at most all services required by u_{ik}. The instances in J_i may differ from u_{ik} for cost and quality attribute (e.g. reliability and response time).
3. ***Replacing existing unit instances with functionally equivalent ones developed in-house*:** An existing instance of a software unit u_i may be replaced with one developed in-house. Developers could opt for different building strategies resulting in different in-house instances, i.e., the elements of the set $\overline{J_i}$. The values of quality attributes of such optional instances (e.g., reliability, response time) could vary due to the values of the development process parameters (e.g. experience and skills of the developing team).
4. ***Modifying the interactions among software units in a certain functionality:*** The system dynamics may be modified by introducing/removing interactions among software units within a certain functionality.

Optimization model formulation.

Model Variables. Let x_{ij} $(1 \leq i \leq n,\ j \in J_i \cup \overline{J_i})$ be the binary variable that is equal to 1 if the instance j is chosen for the software unit i, and 0 otherwise. Moreover, let z_h $(1 \leq h \leq |NewS|)$ be the binary variable that is equal to 1 if the new software units $h$is chosen and 0 otherwise.

Let us analyze the system on the base of p quality attributes (such as cost, response time, availability, etc.). Suppose moreover that each attribute of any software unit depends on the value of parameters α_i^k's, β_i^k's, and γ_{ij}^k 's, where (i) the vector α_i^k describes the (at most) u software architecture observable parameters, e.g., the average number of invocations of a software unit within the execution scenarios considered for the software architecture, (ii) the vector β_i^k contains the (at most) v hardware observable parameters, e.g., the processing capacity of the node hosting the software unit, that is measured, for example, as the average number of instructions per second that the resource can execute, and (iii) the vector γ_{ij}^k represents the (at most) w features of the implementation of u_i, e.g., the reliability of the instance used for replacing the existing unit. For the k quality attributes of a provided instance, the value of the features γ_{ij}^k's is assumed to be either given from the software unit provider or estimated from the customer.

[12] Notice that such type of action has to be associated to another action that indicates how this unit interacts with existing units, therefore it modifies the interactions within certain functionalities (see last type of software action).

On the contrary, for an in-house developed instance the γ_{ij}^k 's can be predicted by considering variables of the decision planning.

Let $\Gamma_k : \mathbb{R}^u \times \mathbb{R}^v \times \mathbb{R}^w \rightarrow \mathbb{R}$ ($\bar{\Gamma}_k : \mathbb{R}^u \times \mathbb{R}^v \times \mathbb{R}^w \rightarrow \mathbb{R}$) be the function that, on the base of the above parameters, returns the value of the k-th quality attribute ($1 \leq k \leq p$) of an existing (new) software unit. In particular, let $\Lambda_{ij}^k = \Gamma_k(\alpha_i^k, \beta_i^k, \gamma_{ij}^k)$ the value of the k-th attribute of the provided/in-house instance u_{ij}.

We can represent the value of the k-th quality attribute of the i-th existing software unit as a function of the decisional strategy **x**:

$$\theta_i^k = \sum_{j \in \bar{J}_i \cup J_i} \Lambda_{ij}^k \, x_{ij} \qquad (1)$$

Similarly, we can represent the value of the k-th quality attribute of the h-th new software unit as a function of the decisional strategy **z**:

$$\bar{\theta}_h^k = z_h \bar{\Gamma}_k \left(\alpha_i^k, \beta_i^k, \gamma_{ij}^k\right) \qquad (2)$$

Let G_k: $\mathbb{R}^n \times \mathbb{R}^{|NewS|} \rightarrow \mathbb{R}$, with $(1 \leq k \leq p)$, be the function that returns the k-th quality attribute of the whole system on the base of the same attributes of each existing/new software unit. And let us assume (without loss of generality) that the values of each quality attribute k are constrained strained to be above a lower threshold value Θ^k. Assume, moreover, that the cost is the first quality attribute, i.e., θ_i^0 ($\bar{\theta}_i^0$) express the cost of the existing (new) software units. Finally, let $Cost$: $\mathbb{R}^n \times \mathbb{R}^{|NewS|} \rightarrow \mathbb{R}$ be the cost function of the whole system that clearly depends on the costs of all the existing (new) software units. Different cost models could be used to define $Cost$, e.g., it may also include the potential costs of software unit adaption (i.e. the glueware). The general formulation of the optimization model for the QoS tradeoffs analysis is given by:

$$\min_{\mathbf{x},\mathbf{z}} Cost(\theta^0, \bar{\theta}^0) \qquad (3)$$

$$G_k(\theta^0, \bar{\theta}^0) \geq \Theta^k \qquad \forall k = 1 \dots p$$

$$\sum_{j \in \bar{J}_i \cup J_i} \Lambda_{ij}^k \, x_{ij} = \theta_i^k \qquad \forall k = 1 \dots p, \forall i = 1 \dots n$$

$$z_h \bar{\Gamma}_k \left(\alpha_h^k, \beta_h^k, \gamma_h^k\right) = \bar{\theta}_h^k \qquad \forall k = 1 \dots p, \forall h = 1 \dots |NewS|$$

$$x_{ij} \in \{0,1\} \qquad \forall i = 1 \dots n, \forall j$$
$$= 1 \dots p \sum_{j \in \bar{J}_i \cup J_i} x_{ij} = 1 \qquad \forall i = 1 \dots n$$

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

Other constraints (e.g., equations to predict α_i^k's and β_i^k's)

6.2 THE SHEPHERD FRAMEWORK

In this section, we provide an overview of the SHEPhERd framework [103], which we have introduced in the context of component-based architectures.

Figure 36 shows the SHEPhERd framework within its working environment. The framework basically comprises two modules: a *Model builder* and a *Model solver*.

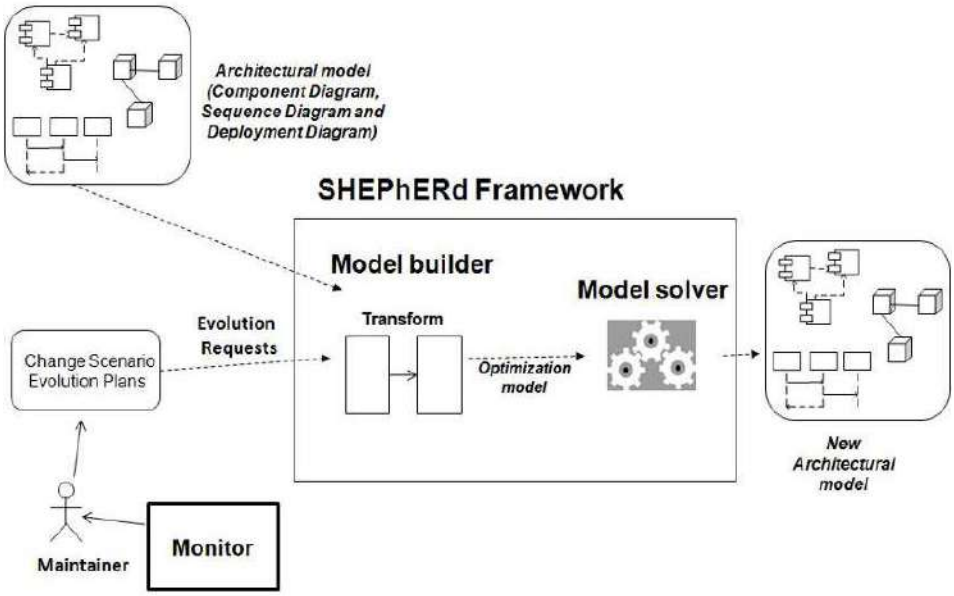


FIGURE 36: *THE SHEPHERD FRAMEWORK AND ITS ENVIRONMENT*

SHEPhERd framework Input. A *primary input* to the framework 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, and (iii) a Deployment Diagram describing the platform architecture.

The system maintainer, through a Monitor module, is able to perceive non-functional requirement violations in the runtime system. She/he defines evolution plans for new and/or violated requirements that represent change scenarios. After receiving an evolution request from the system maintainer, the Model builder generates the optimization model in the format accepted by a solver (e.g., LINGO[13] that we have used in [103]).

The *Model builder* first allows users to annotate the UML diagrams with additional data that represent the optimization model parameters, such as failure probabilities of software components, or the processing capacity of the platform nodes. Then, it transforms the annotated model into an optimization model in the format accepted from the solver.

SHEPhERd framework Output. The optimization model is processed by the Model solver, which produces the results, which consist of a set of evolution actions. It suggests how to adapt both the static and dynamic aspects of the software architecture. Moreover, the platform architecture is modified by re-deploying existing components and/or deploying new components on the existing nodes.

A new software architecture is obtained by modifying its structure and behavior. To modify the structure, our approach suggests replacing existing components with different available instances and/or to introduce new components into the system. With regard to the system behavior, the model is focused on the system scenarios (expressed, for example, as UML Sequence Diagrams) by removing or introducing interaction(s) between existing or new components. The platform architecture (modeled, for example, with an UML deployment diagram) can also be modified by re-deploying existing components and/or deploying new components.

In [103], the mathematical formulation of the optimization model that SHEPhERd generates and solves can be found. Details on model formulation can be found in [103].

The goal of our optimization model is to find the optimal set of actions needed to tackle required changes to the software architecture. "Optimal" here denotes actions that incur minimum cost while guaranteeing a certain level of reliability and performance.
The objective function under the main reliability and performance constraints, plus the constraints on the model variables, represents our optimization model. The model solution determines the evolution plan to choose for each change requirement, in order to minimize the software evolution costs under the reliability and performance constraints.

[13] [Online]. Available: www.lindo.com.

6.3 THE SAQO (SYSTEM ADAPTATION WITH QUALITY OPTIMIZATION) FRAMEWORK

In this section, we introduce the SAQO (System Adaptation with Quality Optimization) framework, which extends the SHEPhERd framework. Figure 24 shows the SAQO framework within its working environment.

The framework SAQO allows storing the specification of requirements, architectural decisions, and their interactions in a repository. The internal structure of the repository is compliant with the metamodel in Figure *25*.

SAQO is a complex specification environment adopting the metamodel for the adaptation space. SAQO allows to:

- Support the software architects/maintainers to maintain the interactions and conflicts between requirements, between design decisions, and between requirements and design decisions. The support includes automatic detection (by model checking techniques) of interactions and conflicts mostly in the part of the architecture design decisions and propagation of interaction between different levels.
- Automatically produce the space of possible feasible architectural solutions obtained by instantiating parametric design decisions. Each solution is computed taken into account the specification constraints associated with the design decisions and the known interactions and conflicts between concrete design options.
- Dynamically adapt a service-based system in an automated manner. SOQA is based on an optimization model that allows to choose among the possible solutions (produced in the previous point) the concrete solutions that minimizes cost, while keeping system qualities (e.g., the reliability and the response time) within certain thresholds.

For example, SOQA can be used to suggest the "best" actions to be taken upon a certain change scenario arising. A change scenario is a set of new requirements that induce changes in the structural and behavioral aspects of the software architecture. A new software architecture is obtained by modifying its structure and behavior. To modify the structure, SOQA suggests replacing existing elementary services with different available instances and/or to introduce new services into the system. With respect to the changes in the system behavior, it modifies the architectural design decisions (represented as parametric BPEL processes) by removing or introducing interactions between existing services and/or between existing and new services. The parametric design decision is instantiated in order to have a space of feasible concrete design decisions and the best concrete design decision resulting from the optimization phase is suggested.

SAQO framework Input. As shown in Figure 37, the input of our framework is a parametric BPEL which represents an architectural decision, which has to be concretized by instantiating the parameters with concrete adaptation decisions. The concrete decisions which are candidates for instantiation are retrieved in the repository by exploiting the search expression associated with parameters. The set

of candidates are filtered by using constraints (to be defined), interaction and conflict information. The task is performed by the *Concretization module*.

The *Conflict Analysis* module takes in input a design option and produces an executable specification whose behaviors are checked against invariant and reachability constraints in a model checking environment (e.g., SPIN[14]).

The PROMELA language[15], for example, can be used for the executable specification. Therefore, a BPEL is translated into a PROMELA program and its behaviours are checked against state and reachability properties. Conflicts and interactions detected are stored in the repository, and possibly used for complete the knowledge about interaction and conflicts of stored entities. The output of the *Conflict Analysis* module is the adaptation space, namely a set of feasible design options over which the next step of optimization is taken.

It is a module obtained by integration of the SHEPhERd framework proposed in [103]. Similar to the SHEPhERd framework, the Optimizer module of the SOQA framework comprises two main modules: a Model builder and a Model solver (see previous section for more details)

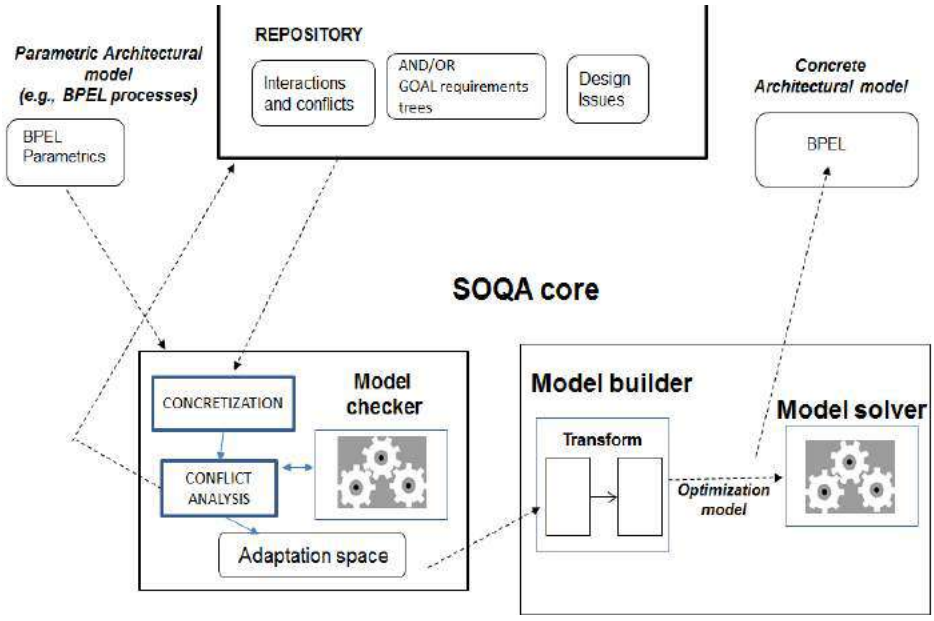


FIGURE 37: *THE SOQA FRAMEWORK AND ITS ENVIRONMENT*

[14] http://spinroot.com/spin/whatispin.html

[15] http://spinroot.com/spin/Man/grammar.html

6.3.1 The Metamodel

In this section, we describe the metamodel for the adaptation space of service based applications (see Figure 38).

The metamodel allows to represent: (a) structured requirements with particular concern on their interactions, conflicts, and conflict resolutions; (b) parametric and concrete structured design decisions associated with the requirements together with interactions and conflicts between design solutions; and (c) transformation of design decisions in order to support the adaptation.

In the following we discuss the main entity of the metamodel related to the (i) *requirement modeling*, and (ii) *design modeling.*

Requirement modeling.

Requirement: A requirement can also be seen as a goal. A goal can be a functional requirement (hard-goal) or non-functional requirement (softgoal). According to [104], goals represent stakeholder intentions, which are manifestations of intent which may or may not be realized. A requirement can be (recursively) structured into AND/OR composition (sub-) requirements defining an AND/OR tree like structure. A requirement has a textual description (e.g., a natural language specification), and a constraint consisting of a formal expression over attribute-value pairs associated with the entity Requirement. A requirement may have a number of associated issues.

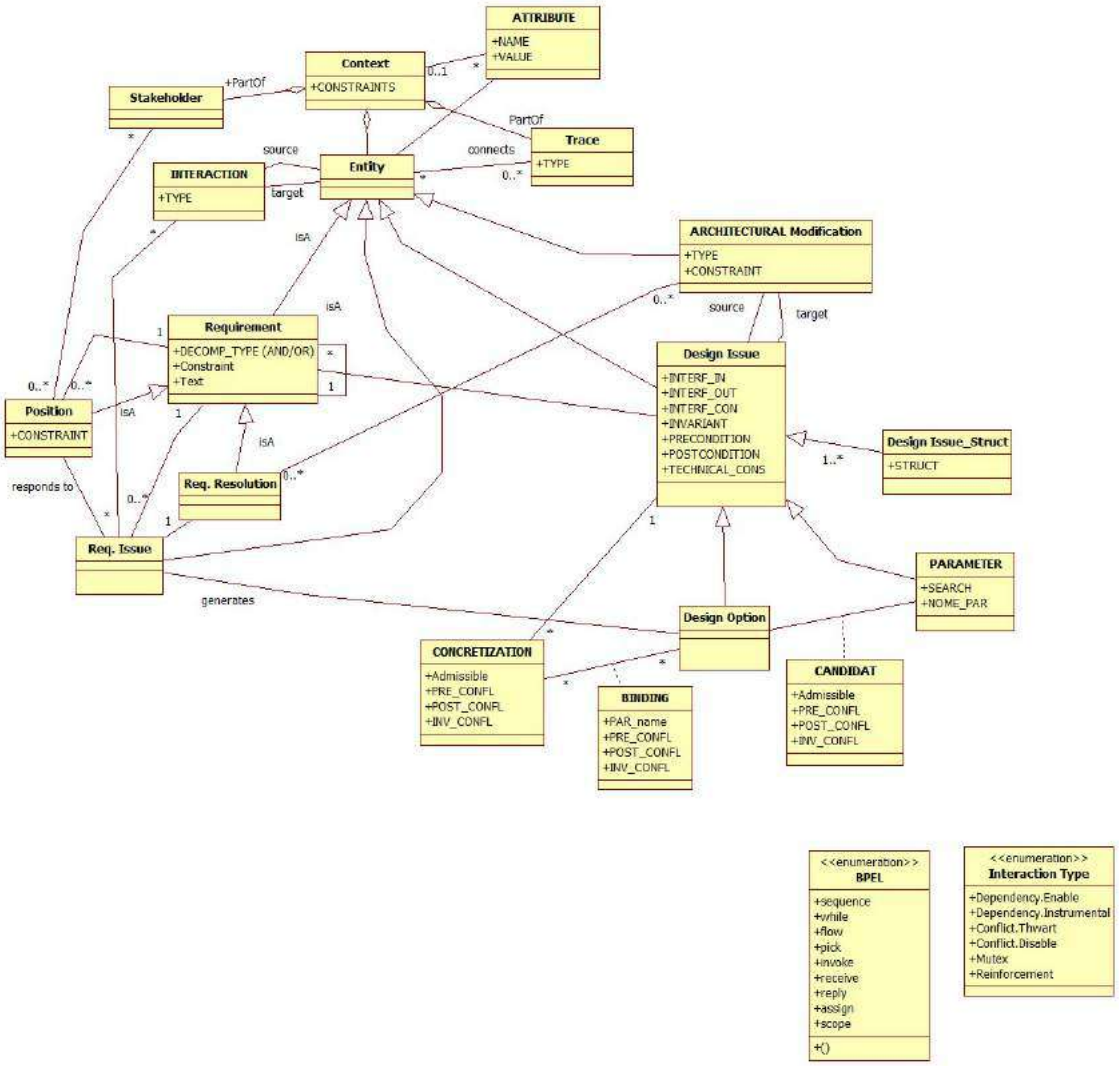


FIGURE 38*: ADAPTATION SPACE EXPLORATION METAMODEL*

Position: For a requirement, the stakeholders may express different positions with respect to an Issue associated with a requirement. A position provides an (alternative) solution of an issue. A position may be in conflict with other positions related to the same issue. A requirement resolution is a requirement which intends to overcome the conflicts to different positions of the same requirement. Issues are questions, such as, "how will requirement R_i be satisfied?", "what does term t_i of R_i mean?". Remark: this part of the model addresses only different interpretation of the same requirement and do not address as in [105] statements of the form "requirements R_i and R_j appear to conflict, how can they be resolved?" [105]. The solution of this problem is given by possibly associating *Requirement Issue* also to an *Interaction* between requirements. In summary, a requirement issue can be opened either for different positions with respect to a requirement or for a conflicting interaction among requirements.

Design Modeling

A design issue represents an architectural schema, which is described by a composite abstract structure, namely a BPEL where parametric services can be invoked. We use the standard control operation: sequence, while, switch, flow, invoke. An invocation can take a composite concrete structure or parameter (abstract).

A design issue has the following attributes:

- *Interface Input*: It is the set of required services. It is given by an ordered set of logical names.
- *Interface Output:* It is the set of provided services. It is given by an ordered set of logical names.
- *Internal Interface Connection*: It is the set of interfaces composition of internal modules. It is given by a set of pairs of the form (M1.Out1, M2.In2) where M1 and M2 are logical names of the modules of the design issue, and Out1 is an interface postcondition of M1 and In2 is an interface precondition of M2. Moreover, we can have pairs of the form (self.In, M1.In1) and (self.Out, M1.Out1) connecting interface post and preconditions of the design issue, respectively, with post and preconditions of an internal node.
- *Precondition:* A constraint which has to be satisfied to activated the solution.
- Postcondition: A constraint which is satisfied at the termination of the execution.
- *Invariant:* A constraint which is satisfied in each intermediate stable state, i.e., before
  and after the execution of each atomic action.
- *Technical Constraints*: Technical limitations, for instance, required technology.
- All the constraints are boolean expression freely constructed with boolean connectives
  over atomic proposition of the form: *EntityName.AttributeName op Value*, with op in $\{>, \leq, \geq, <, =, \neq\}$. Notice that Design Issue inherits from Entity the possibility to associate a set of attributes together with their current values.
- A search attribute in a parameter is a query like string giving the set of design options to be considered for the instantiation of the parameter (the parameter domain). Notice that it is not guaranteed that the design options in the result set are admissible.

A **design option** is described by a composite concrete structure, namely a BPEL which allows only concrete invocations (there is no occurrence of parameters). With reference to the metamodel note that a design option is a special case of a design issue with no-occurrence of parameters. In the metamodel, we have an association which binds the Design option with Parameter. An admissible binding should preserve pre, post, and invariant conditions of Parameters and Design Options.

A **concretization** is a simultaneous binding of all of the Parameters of a Design Issue with a corresponding number of admissible design options. The concretization is admissible if the pre, post, and invariant conditions of the Design Option are fulfilled and if the individual pre, post, and invariant conditions of each Design Option continue to hold when they are placed in the context of the Design Issue.

The pre confl, post confl, inv confl attributes of Concretization report possible conflicts related to a concretization. The contribution to the possible conflict of each parameter binding is reported in the pre confl, post confl, inv confl attributes of the association classes between Concretization and Design Option.

7 CONCLUSIONS

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 approach implemented as an optimization framework for dynamically modeling: (i) fault detection and correction processes of systems functionalities (modules) through the SRGMs that best fit the actual testing data, (ii) testing cost/time constraints, and (iii) parameter-specific uncertainties phenomena. So that the systems functionalities (modules) with shorter time (budget) are tested and that reveled bugs are fixed earlier. We provide guidelines for practitioners. We provide support for their testing allocation decisions based on cost, time, and software quality.

We have also proposed an automatic prioritization approach for large software systems that embeds the "code churn" measure. Specifically, we have provided support for optimizing regression functional testing with coverage and churn metrics. Moreover, our work has been also focused on the automation of the support for the architectural decisions. Specifically, we have focused on the (i) modeling and analysis of QoS tradeoffs of a software architecture based on optimization models, and (ii) definition of framework for supporting the software architects/maintainers. More specifically, we support the software architects/maintainers to maintain the interactions and conflicts between requirements, between design decisions, and between requirements and design decisions. The support includes automatic detection (by model checking techniques) of interactions and conflicts mostly in the part of the architecture design decisions and propagation of interaction between different levels. Our approach also allows producing the space of possible feasible architectural solutions obtained by instantiating parametric design decisions. Each solution is computed taken into account the specification constraints associated with the design decisions and the known interactions and conflicts between concrete design options.

8 REFERENCES

[1] Deliverable D3.1, "First measurement/prediction models-based process*",*
7th Framework Programme IAPP Marie Curie program for project ICEBERG no. 324356, May 2014.

[2] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, "Not going to take this anymore: Multi-objective overtime planning for Software Engineering projects", In Software Engineering (ICSE), 2013 35th International Conference on, May 2013.

[3] Zai, K. Tang, and X. Yao, "Multi-Objective Approaches to Optimal Testing Resource Allocation in Modular Software Systems", Reliability, IEEE Transactions on, 59(3):563–575, 2010.

[4] R.-H. Hou, S.-Y. Kuo, and Y.-P. Chang, "Efficient allocation of testing resources for software module testing based on the hyper-geometric distribution software reliability growth model", In Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on, pages 289–298, 1996.

[5] G. Carrozza, R. Pietrantuono, and S. Russo, "Dynamic test planning: a study in an industrial context", International Journal on Software Tools for Technology Transfer, pages 1–15, 2014.

[6] D. Cotroneo, R. Pietrantuono, and S. Russo, "Testing techniques selection based on ODC fault types and software metrics", Journal of Systems and Software, 86(6):1613–1637, 2013.

[7] P. C. Jha, D. Gupta, B. Yang, and P. K. Kapur. Optimal testing resource allocation during module testing considering cost, testing effort and reliability. Computers & Industrial Engineering, 57(3):1122–1130, 2009.

[8] C.-Y. Huang and M. Lyu. Optimal testing resource allocation, and sensitivity analysis in software development. Reliability, IEEE Transactions on, 54(4):592–603, 2005.

[9] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, and L. Grunske. Model-based performance analysis of software architectures under uncertainty. In QoSA, pages 69–78. ACM, 2013.

[10] I. Meedeniya, A. Aleti, and L. Grunske. Architecture-driven reliability optimization with uncertain model parameters. Journal of Systems and Software, 85(10):2340–2355, 2012.

[11] G. Baio, "Bayesian Methods in Health Economics", Chapman and Hall/CRC, 2012.

[12] The Bugzilla bug tracking tool. [Online]. Available: http://www.bugzilla.org/.

[13] H. Hosseini, R. Nguyen, and M. W. Godfrey, "A Market-Based Bug Allocation Mechanism Using Predictive Bug Lifetimes", In 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012, pages 149–158. IEEE Computer Society, 2012.

[14] H. Okamura, Y. Watanabe, and T. Dohi, "An iterative scheme for maximum likelihood estimation in software reliability modeling", In Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on, pages 246–256, 2003.

[15] H.-G. Beyer and B. Sendhof,. "Robust optimization - a comprehensive survey", Computer Methods in Applied Mechanics and Engineering, 196(33-34):3190 – 3218, 2007.

[16] H. Ziv and D. J. Richardson, "Bayesian-network confirmation of software testing uncertainties", In Proceedings of the Sixth European Software Engineering Conference (ESEC), Zurich, pages 22–25, 1997.

[17] H. Ziv and D. J. Richardson, "Constructing Bayesian-network models of software testing and maintenance uncertainties", In ICSM, pages 100–. IEEE Computer Society, 1997.

[18] K. Yue., "Generating interesting scenarios from system descriptions", In Proceeding of the 1st international conference on Industrial and engineering applications of artificial intelligence and expert systems, pages 212 – 218, 1988.

[19] S. G. Elbaum and D. S. Rosenblum, "Known unknowns: testing in the presence of uncertainty", In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), 2014, pages 833–836. ACM, 2014.

[20] R. Roshandel, N. Medvidovic, and L. Golubchik, "A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level", In QoSA, pages 108–126, 2007.

[21] S. Raychaudhuri, "Introduction to monte carlo simulation", In Winter Simulation Conference, pages 91–100. WSC, 2008.

[22] J. Musa, "Operational profiles in software-reliability engineering", Software, IEEE, 10(2):14–32, Mar 1993.

[23] O. Baysal, M. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs", In Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on, pages 297–298, May 2009.

[24] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In 35th International Conference on Software Engineering, ICSE '13, pages 1042–1051. IEEE / ACM, 2013.

[25] H.-W. Jung and B. Choi. Optimization models for quality and cost of modular software systems. European Journal of Operational Research, 112(3):613 – 619, 1999.

[26] C.-Y. Huang and M. Lyu, "Optimal release time for software systems considering cost, testing-effort, and test efficiency", Reliability, IEEE Transactions on, 54(4):583–591, 2005.

[27] C.-Y. Huang, S.-Y. Luo, and M. Lyu, "Optimal software release policy based on cost and reliability with testing efficiency", In Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International, pages 468–473, 1999.

[28] C.-Y. Huang, J.-H. Lo, S.-Y. Kuo, and M. Lyu, "Software reliability modeling and cost estimation incorporating testing-effort and efficiency", In Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on, pages 62–72, 1999.

[29] S. Yamada, J. Hishitani, and S. Osaki, "Software-reliability growth with a Weibull test-effort: a model and application", Reliability, IEEE Transactions on, 42(1):100–106, Mar 1993.

[30] C. Huang and J. Lo, "Optimal resource allocation for cost and reliability of modular software systems in the testing phase", Journal of Systems and Software, 79(5):653–664, 2006.

[31] B. W. Boehm, "Software Engineering Economics", Prentice Hall PTR, 1st edition, 1981.

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

[33] V. Cortellessa, F. Marinelli, R. Mirandola, and P. Potena, "Quantifying the influence of failure repair/mitigation costs on service-based systems", In IEEE 24th International Symposium on Software Reliability Engineering, ISSRE, pages 90–99. IEEE, 2013.

[34] H. Hemmati, M. Nagappan, and A. E. Hassan, "Investigating the effect of "defect co-fix" on quality assurance resource allocation: A search-based approach", Journal of Systems and Software, (0):–, 2014.

[35] J. Ren, M. Harman, and M. D. Penta, "Cooperative Co-evolutionary Optimization of Software Project Staff Assignments and Job Scheduling", Search Based Software Engineering - Third International Symposium, SSBSE 2011. Proceedings, volume 6956 of LNCS, pages 127–141. Springer, 2011.

[36] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, "Not going to take this anymore: Multi-objective overtime planning for Software Engineering projects", In Software Engineering (ICSE), 2013.

[37] J. J. Durillo, A. J. Nebro, and E. Alba, "The jMetal framework for multi-objective optimization: Design and architecture", In IEEE Congress on Evolutionary Computation, pages 1–8. IEEE, 2010.

[38] C.-Y. Huang, S.-Y. Kuo, and M. R. Lyu, "An Assessment of Testing-Effort Dependent Software Reliability Growth Models", IEEE Transactions on Reliability, 56(2):198–211, 2007.

[39] F. Parr, "An Alternative to the Rayleigh Curve Model for Software Development Effort", Software Engineering, IEEE Transactions on, SE-6(3):291–296, May 1980.

[40] P. Kapur, H. Pham, S. Anand, and K. Yadav, "A Unified Approach for Developing Software Reliability Growth Models in the Presence of Imperfect Debugging and Error Generation", Reliability, IEEE Transactions on, 60(1):331–340, March 2011.

[41] X. Zhang, X. Teng, and H. Pham, "Considering fault removal efficiency in software reliability Assessment", Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 33(1):114–120, Jan 2003.

[42] R. Peng, Y. Li, W. Zhang, and Q. Hu, "Testing effort dependent software reliability model for imperfect debugging process considering both detection and correction", Reliability Engineering & System Safety, 126(0):37 – 43, 2014.

[43] Y. S. Dai, M. Xie, K. L. Poh, and B. Yang, "Optimal Testing-resource Allocation with Genetic Algorithm for Modular Software Systems", J. Syst. Softw., 66(1):47–55, Apr. 2003.

[44] C. Stringfellow and A. A. Andrews, "An Empirical Method for Selecting Software Reliability Growth Models", Empirical Softw. Engg., 7(4):319–343, Dec. 2002.

[45] K. Sharma, R. Garg, C. Nagpal, and R. K. Garg, "Selection of Optimal Software Reliability Growth Models Using a Distance Based Approach", Reliability, IEEE Transactions on, 59(2):266–276, June 2010.

[46] N. Ullah, M. Morisio, and A. Vetro, "A Comparative Analysis of Software Reliability Growth Models using Defects Data of Closed and Open Source Software", In Software Engineering Workshop (SEW), 2012 35th Annual IEEE, pages 187–192, Oct 2012.

[47] M. Lyu, "Software Reliability Engineering: A Roadmap. In Future of Software Engineering", FOSE '07, pages 153–170, May 2007.

[48] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, F. Törner, W. Meding, and C. Höglund, "Selecting software reliability growth models and improving their predictive accuracy using historical projects data", Journal of Systems and Software, 98(0):59 – 78, 2014.

[49] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Torner, "Evaluating long-term predictive power of standard reliability growth models on automotive systems", In Software Reliability Engineering (ISSRE), IEEE 24th International Symposium on, pages 228–237, Nov 2013.

[50] H. Pham, "Software reliability and cost models: Perspectives, comparison, and practice", European Journal of Operational Research, 149(3):475 – 489, 2003.

[51] G. Canfora and L. Cerulo. How software repositories can help in resolving a new change request. In In Workshop on Empirical Studies in Reverse Engineering, 2005.

[52] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?", In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 361–370. ACM, 2006.

[53] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers", In Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), pages 131–140. IEEE, 2009.

[54] O. Baysal, M. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs", In Program Comprehension, ICPC '09. IEEE 17th International Conference on, pages 297–298, 2009.

[55] J. Xiao and W. Afzal, "Search-based resource scheduling for bug fixing tasks", In Search Based Software Engineering (SSBSE), 2010 Second International Symposium on, pages 133–142, Sept 2010.

[56] G. Antoniol, M. Di Penta, and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project", In Software

Maintenance, ICSM'05. Proceedings of the 21st IEEE International Conference on, pages 240–249, 2005.

[57] N. Kaushik, M. Amoui, L. Tahvildari,W. Liu, and S. Li, "Defect Prioritization in the Software Industry: Challenges and Opportunities", In Software Testing, Verification and Validation (ICST), IEEE Sixth International Conference on, pages 70–73, 2013.

[58] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "MIMIC: locating and understanding bugs by analyzing mimicked executions", In ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 815–826. ACM, 2014.

[59] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs", In the 17th IEEE International Conference on Program Comprehension, ICPC, pages 70–79. IEEE Computer Society, 2009.

[60] N.Wattanapongskorn and D.W. Coit, "Fault-tolerant embedded system design and optimization considering reliability estimation uncertainty", Reliability Engineering & System Safety, 92(4):395 – 407, 2007.

[61] D. Doran, M. Tran, L. Fiondella, and S. S. Gokhale, "Architecture-based Reliability Analysis With Uncertain Parameters", In SEKE'11, pages 629–634, 2011.

[62] N. Esfahani, K. Razavi, and S. Male, "Dealing with Uncertainty in Early Software Architecture", In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 21:1–21:4. ACM, 2012.

[63] B. Pachauri, A. Kumar, and J. Dhar, "Modeling optimal release policy under fuzzy paradigm in imperfect debugging environment", Information and Software Technology, 55(11):1974 –1980, 2013.

[64] C. Huang and J. Lo, "Optimal resource allocation for cost and reliability of modular software systems in the testing phase", Journal of Systems and Software, 79(5):653–664, 2006.

[65] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software", In Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01, pages 312–326. ACM, 2001

[66] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey", Softw. Test., Verif. Reliab., 22(2):67–120, 2012.

[67] Z. Anwar and A. Ahsan, "Exploration and Analysis of Regression Test Suite Optimization", SIGSOFT Softw. Eng. Notes, 39(1):1–5, Feb. 2014.

[68] C.-T. Lin, K.-W. Tang, C.-D. Chen, and G. Kapfhammer, "tReducing the Cost of Regression Testing by Identifying Irreplaceable Test Cases", In Genetic and Evolutionary Computing (ICGEC), 2012 Sixth International Conference on, pages 257–260, Aug 2012.

[69] M. J. Harrold, D. S. Rosenblum, G. Rothermel, and E. J. Weyuker, "Empirical Studies of a Prediction Model for Regression Test Selection", IEEE Trans. Software Eng., 27(3):248–263, 2001.

[70] A. Nanda, S. Mani, S. Sinha, M. Harrold, and A. Orso, "Regression testing in the presence of non-code changes", In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pages 21–30, March 2011.

[71] J. Zheng, L.Williams, and B. Robinson, "Pallino: automation to support regression test selection for cots-based applications", In 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA, pages 224– 233. ACM, 2007.

[72] N. Kaushik, M. Salehie, L. Tahvildari, S. Li, and M. Moore, "Dynamic Prioritization in Regression Testing", In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pages 135–138, 2011.

[73] A. Bertolino, P. Inverardi, and H. Muccini, "Software architecture-based analysis and testing: a look into achievements and future challenges", Computing, 95(8):633–648, 2013.

[74] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation", J. Syst. Softw., 86(8):1978–2001, Aug. 2013.

[75] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing", IEEE Trans. Software Eng., 27(10):929–948, 2001.

[76] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies", IEEE Trans. Software Eng., 28(2):159–182, 2002.

[77] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing", In ISSTA, pages 102–112, 2000.

[78] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study", In Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on, pages 179–188, 1999.

[79] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. Proceedings of ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada, pages 329–338. IEEE Computer Society, 2001.

[80] H. Srikanth, S. Banerjee, L. Williams, and J. A. Osborne, "Towards the prioritization of system test cases", Softw. Test., Verif. Reliab., 24(4):320–337, 2014.

[81] J.-M. Kim and A. Porter, "A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments", In Proceedings of ICSE '02, pages 119–129. ACM, 2002.

[82] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "TimeAware Test Suite Prioritization". In Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA '06, pages 1–12. ACM, 2006.

[83] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing", Journal of Systems and Software, 85(3):626 – 637, 2012.

[84] Z. Li, M. Harman, and R. Hierons. "Search Algorithms for Regression Test Case Prioritization. Software Engineering", IEEE Transactions on, 33(4):225–237, April 2007.

[85] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A Unified Test Case Prioritization Approach", ACM Trans. Softw. Eng. Methodol., 24(2):10:1–10:31, Dec. 2014.

[86] J. Jasz, L. Lango, T. Gyimothy, T. Gergely, A. Beszedes, and L. Schrettner., "Code Coveragebased Regression Test Selection and Prioritization in WebKit", In Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), ICSM '12, pages 46–55. IEEE Computer Society, 2012.

[87] A. B. Sánchez, S. Segura, and A. R. Cortés, "A Comparison of Test Case Prioritization Criteria for Software Product Lines", In Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, pages 41–50. IEEE Computer Society, 2014.

[88] J. Ouriques, E. Cartaxo, and P. Machado, "On the Influence of Model Structure and Test Case Profile on the Prioritization of Test Cases in the Context of Model-Based Testing", In Software Engineering (SBES), 2013 27th Brazilian Symposium on, pages 119–128, Oct 2013.

[89] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In ICSE, 2015.

[90] S. G. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments", In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), pages 235–245. ACM, 2014.

[91] The eXist-db database. [Online]. Available: http://existdb.org/exist/apps/homepage/index.html.

[92] The JaCoCo tool. [Online]. Available: http://www.eclemma.org/jacoco/.

[93] The Code Churn tool. [Online]. Available: http://www.assioma.net/.

[94] The Sonar tool. [Online]. Available: http://www.sonarqube.org/.

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

[96] R. Lingampally, A. Gupta, and P. Jalote. A Multipurpose Code Coverage Tool for Java. In System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, pages 261b–261b, Jan 2007.

[97] The RaptorXML tool. [Online]. Available: http://www.altova.com/raptorxml.html.

[98] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz,, L. Lango, T. Gyimothy, "Code coverage-based regression test selection and prioritization in WebKit," Software Maintenance (ICSM), 2012 28th IEEE International Conference on , vol., no., pp.46,55, 23-28, 2012.

[99] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact", Empirical Software Engineering: An International Journal, 10(4):405–435, 2005.

[100] A. Carzaniga, D. S. Rosenblum, and A. L.Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service", In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, pages 219–227, New York, NY, USA, 2000. ACM.

[101] V. Cortellessa and P. Potena, "Supporting architectural decisions through software quality optimization models", Tutorial at 25th IEEE International Symposium on Software Reliability Engineering (ISSRE) November 3-6, 2014 (http://issre.net/tutorials).

[102] P. Potena, I. Crnkovic, F. Marinelli, and V. Cortellessa, "Software Architecture Quality of Service Analysis based on Optimization Models", Chapter in Intelligent Decision Making in Quality Management, Springer (Accepted for pubblication).

[103] V. Cortellessa, R. Mirandola, and P. Potena, "Managing the evolution of a software architecture at minimal cost under performance and reliability constraints", Science of Computer Programming (Elsevier), 98: 439-463 (2015). DOI: 10.1016/j.scico.2014.06.001.

[104] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. do Prado Leite, "From goals to high-variability software design", In ISMIS, volume 4994 of LNCS, pages 1–16. Springer, 2008.

[105] W. N. Robinson and S. Volkov, "Conflict-Oriented Requirements Restructuring", In GSU CIS Working Paper 96-15, Georgia State University, Atlanta, GA, 1996.

[106] S. Yamada, T. Ichimori, M. Nishiwaki: Optimal Alloca- tion Policies for Testing-Resource Based on a Software Reliability Growth Model. Int. Journal of Mathematical and Computer Modeling. 22(10-12), 295-301 (1995)

[107] M.R. Lyu, S. Rangarajan, A.P.A. van Moorsel: Opti- mal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development. IEEE Trans- actions on Reliability, 51 (2), 336-347 (2002)

[108] C.Y. Huang, J.H. Lo, S.Y. Kuo, M.R. Lyu: Optimal Al- location of Testing Resources for Modular Software Sys- tems. In: Proc. 13th Int. Symposium on Software Relia- bility Engineering (ISSRE), pp. 129-138 (2002)

[109] I. Meedeniya, A. Aleti, and L. Grunske. Architecture-driven reliability optimization with uncertain model parameters. Journal of Systems and Software, 85(10):2340–2355, 2012.

[110] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, and L. Grunske. Model-based perfor- mance analysis of software architectures under uncertainty. In QoSA, pages 69–78. ACM, 2013.

[111] N. F. Schneidewind, "Modelling the fault correction process," in Pro- ceedings of the 12th International Symposium on Software Reliability Engineering, 2001, pp. 185–190.

[112] R. Rubinstein and D. Kroese. Simulation and the Monte Carlo method. Wiley-interscience, 2008.

[113] G. Carrozza, R. Pietrantuono, and S. Russo, "Defect analysis in mission- critical software systems: a detailed investigation," J. Softw. Evol. and Proc., vol. 27, no. 1, pp. 22–49, 2014.

[114] R. Pietrantuono, S. Russo, and K. Trivedi, "Software reliability and testing time allocation: An architecture-based approach," *Software Engi- neering, IEEE Transactions on*, vol. 36, no. 3, pp. 323–337, May 2010.